From Simulation to Reality: Migration of Humanoid Robot Control

D i s s e r t a t i o n

zur Erlangung des akademischen Grades
Doktor rerum naturalium
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
der Humboldt-Universität zu Berlin

von
M.sc. Yuan Xu

Präsident der Humboldt-Universität zu Berlin
Prof. Dr. Jan-Hendrik Olbertz

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II
Prof. Dr. Elmar Kulke

Gutachter/Gutachterin
1. Prof. Dr. Hans-Dieter Burkhard
2. Prof. Dr. Verena Hafner
3. Prof. Dr. Yingzi Tan

Tag der Verteidigung: 17. März 2014

Ich widme diese Arbeit
meiner Familie und meinen Freunden

Abstract

Physical simulation is an effective and practical method, to apply to the study and exploration of real world problems. However, simulation can offer valuable results for robotics only in close connection to real robots.

In this thesis, we investigated how to create a mechanism that provides a smooth gradient to transfer humanoid robot control from simulated robot to real robot. We developed a framework for running robots both in real and simulated settings; and evaluated a humanoid robot simulator at a conceptual model level and results level by conducting experiments. Then, we improved the simulator by adding missing models and optimizing parameters with Evolutionary Algorithms. Finally, we developed motions in the simulations, with the help of Machine Learning, and transferred them to real robots successfully.

As a result, a robot team can play soccer using identical controls in both the simulation and real RoboCup leagues. This constitutes a close connection between the communities working with simulated and real robots.

Zusammenfassung

Physikalische Simulation ist eine effektive und praktische Methode, um die Probleme der realen Welt zu untersuchen und zu erforschen. Jedoch kann die Simulation wertvolle Ergebnisse für die Robotik nur in enger Verbindung zu den realen Robotern liefern.

In der Arbeit haben wie Methoden untersucht, die einen glatten Übergang von simulierten zu realen Robotern für die Steuerung humanoider Roboter erlauben. Wir haben ein Framework entwickelt, in dem Roboter sowohl in realen als auch in simulierten Umgebungen arbeiten können. Wir haben einen Simulator für humanoide Roboter auf konzeptioneller und experimenteller Ebene durch entsprechende Experimente evaluiert. Weiterhin haben wir den Simulator um zusätzliche Modelle erweitert und Parameter mithilfe Evolutionärer Algorithmen optimiert. Schließlich haben wir Bewegungen in Simulationen mit Maschinellem Lernen entwickelt und erfolgreich auf reale Roboter übertragen.

Als Resultat können Roboter Teams sowohl in den Simulationsligen als auch in den realen Ligen des RoboCup mit identischen Steuerungen Fußball spielen. Das ergibt eine enge Verbindung zwischen den Entwicklern von simulierten und realen Robotern.

Contents

1 Introduction

> Nothing great was ever achieved without enthusiasm.
>
> — R.W. Emerson

1.1 Backgrounds

There will be a robot in every home in the future, as predicted by experts, e.g. Gates [28]; we will see robots doing our housework and carrying out other tasks in the physical world. Researchers have developed a large number of robots in the last few years. Enthusiasts can buy personal robots or build their own. However, there are challenges in going from science fiction to reality. For example, in order for a robot team to actually play a soccer game against humans [71], several technologies must be incorporated including: sensing the environment, dynamic motions for the humanoid robots, multi-robots collaboration, strategy acquisition, and real-time reasoning.

The development on real robots may be severely limited by the constrained resources. This is especially true in the research of multi-robots systems in areas such as coordination techniques, opponent modeling and machine learning. Traditionally, some of these problems were addressed by using a simulation environment for algorithm development and testing.

1.1.1 Modeling and Simulation

Simulation is the imitation of real systems, state of affairs, or processes. The act of simulating something generally entails representing certain key characteristics or behaviours of a selected physical or abstract system. Representations in simulation are usually called models. They seek to represent empirical objects, phenomena, and physical processes in a logical and objective way. Building and disputing models is fundamental to the scientific enterprise. However, all models in simulation are simplified reflections of reality. Complete and true representation is impossible. Scientific debate often concerns the better model for a given task, such as the most accurate climate model for seasonal forecasting.

Simulation has successful applications in research and industry, e.g. aerospace and nuclear. Simulation has benefits that include:

- Reduced competition for scarce hardware resources;
- No risk to people or equipment;

- Debugging directly;
- The ability to use hardware modules before acquiring them;
- The ability to do repeated tests (for machine learning);
- Evaluating different alternatives during the design phase.

1.1.2 Robotics Simulation

Numerous simulators were developed over the years to assist in the development, testing, and evaluation of robots. The ideal robot development scenario is developing algorithms in the simulation first, and then moving them to real robots without any change. Researchers have developed different approaches to create and integrate robot models and virtual environments. Recently, robot simulation environments have attempted to leverage existing technologies to make a general purpose environment that is capable of simulating the complexities of multi robots in three dimensional settings.

Below we give a short overview of current related works on robot simulators with dynamics simulation for three dimensional environments and with relevance to the humanoid robot.

SimSpark [45, 65] is a generic simulator; and it is the official RoboCup soccer simulator. To create specific simulations, modeling parts of the simulator are built as plug-ins. The plug-ins can be exchanged at run-time for different simulations. In the RoboCup soccer simulation league, the humanoid robot – NAO – is simulated. It is still under active development today. See section 1.1.3 for more details.

SimRobot [48] is another simulator able to replicate arbitrary user-defined robots in three-dimensional space. It includes a physical model which is based on rigid body dynamics. To allow extensive flexibility in building accurate models, different generic bodies, sensors and actuators were implemented. The simulator follows a user-oriented approach by including mechanisms for visualization, direct actuator manipulation, and interaction with the simulated world. SimRobot has been used by the B-Human to simulate the robots and the environment of the Standard Platform League. It was integrated into the environment of Small Size League team B-Smart.

Another general three dimensional multi-robots simulator with graphical interface and dynamics simulation is **Gazebo** [44], which is a part of the **Player/Stage** project [29]. This simulator has a large variety of sensors and comes with models of existing robots such as the Pioneer2DX or the SegwayRMP. The Player server can control the robots and sensors; controllers can be written as a library provided with the simulator. The simulated environment is described in XML files using predefined elements and robots. It also offers the possibility of creating and integrating robots as plug-ins, but this has to be done by code-based modeling in C.

USARSim [73] is a high fidelity simulation of *urban search and rescue* (USAR) robots and environments intended as a research tool for the study of *human-robot interaction* (HRI) and *multi-robot coordination*. It is used in the RoboCup rescue simulation league. The USAR task focuses on robot behaviors, and physical interaction with rubble filled

environments. USARSim supports HRI by accurately rendering user interface elements, accurately representing robot behavior, and accurately representing the remote environment that links the operator's awareness with the robot's behaviors.

Some simulators focus on the humanoid simulation for the complexities of the humanoid robot. **OpenHRP**(Open Architecture Humanoid Robotics Platform) [41] is a virtual humanoid robot platform for the HRP series humanoid robot. It consists of a simulator of humanoid robots and a motion control library for them, which can also be applied to a compatible humanoid robot as it is. It explores humanoid robotics on open software and hardware. It consists of a dynamics simulator, view simulator, motion controllers and motion planners.

Hein [34, 35] developed a simulator named **Simloid** for exploring biped motion control techniques. It is a physical simulation of 19 degrees of freedom; a real humanoid robot model. It has two different approaches to biped motion generation which specify target angle functions for all driven joints. Both approaches were developed and implemented within the physical simulation of the biped robot. The motion parameters were identified and optimized by using evolutionary algorithms, and walking patterns were generated.

Some simulators are designed only for special robot tasks. **OpenGRASP** [49] is an open source simulation toolkit for grasping and dexterous manipulation. It is based on OpenRAVE [19] modular architecture which supports the creation and addition of new functionality and the integration of existing technologies.

Commercial simulation software is also available. **Webots** [16] is a mobile robotics simulation software developed by Cyberbotics Ltd. This software already has many models of existing robots, such as wheeled robots and humanoid robots [14, 55]. The provided robot libraries enable transfer of control programs to commercially available real mobile robots. In [16] Cyberbotics claims that Webots is used by over 750 universities and research centers worldwide.

Microsoft Robotics Studio(MSRS) [53] is a Windows-based environment for academic, hobbyist and commercial developers to easily create robotics applications across a variety of hardware. It can be used in a variety of advanced scenarios with high demands for fidelity, visualization, and scaling. A novice user can use simulation without coding experience and develop in a game-like environment. More than 30 companies have joined the Microsoft Robotics Studio program.

V-REP - the Virtual Robot Experimentation Platform [26] - is a 3D robot simulator, with an integrated development environment. It is based on a distributed control architecture. Control programs (or scripts) can be directly attached to scene objects and run simultaneously in a threaded or non-threaded fashion. An integrated Lua script interpreter makes V-REP versatile. It gives the user the freedom to combine the low/high-level functionalities to obtain new high-level functionalities.

NAOsim [2] is a simulator developed for NAO robots especially. It embeds the middleware of real robots and therefore can be used with other tools developed by Aldebaran. For example, NAOqi C++ modules can run in NAOsim and are connected by their development tool, Choregraphe.

In addition to the simulators described above, another related project is **Physical Vi-**

Table 1.1: Main features of some available three dimensional physic simulators for robots.

Simulator	Physic Engine	Configuration	Architecture	Robots
SimSpark	ODE	Ruby	TCP/IP	Sphere, Humanoid
SimRobot	ODE	XML	Monolithic	Dog, Vehicle
Gazebo	own	XML	TCP/IP, CORBA	Vehicle
USARSim	Unreal	Unreal map	TCP/IP	Vehicle, ...
OpenHRP	own	VRML	CORBA	Humanoid
Simloid	ODE	-	Monolithic	Humanoid
Webots	ODE	VRML	TCP/IP	Vehicle, Humanoid
MSRS	PhysX	XML	DSS	Arm, Vehicle, Dog
NAOSim			NAOqi	NAO
OpenGRASP	Bullet/ODE	COLLADA	Monolithic	Arm, Hand
V-REP	Bullet/ODE	Lau	Monolithic	Vehicle, Humanoid, ...

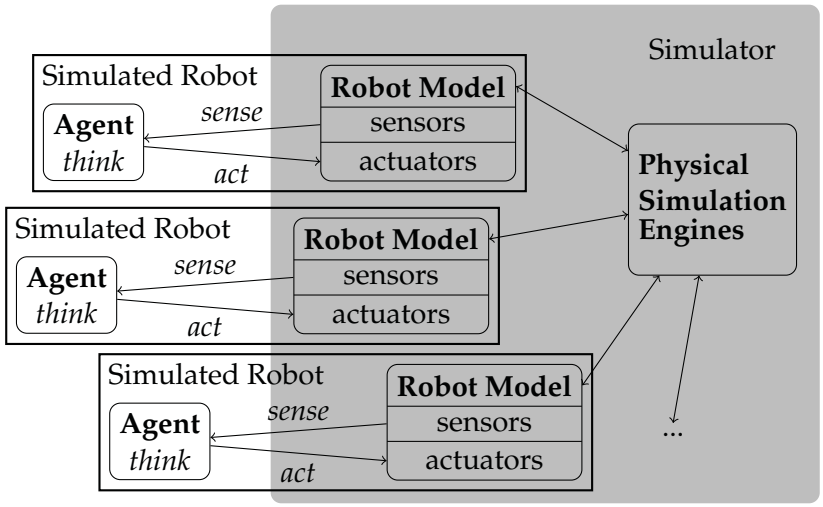


Figure 1.1: Typical architecture of a robot simulator. The robot models, including sensors and actuators, are developed in a simulator. The control programs are developed separately.

sualization [33]. It is based on a miniature multi-robot system which mixes reality and simulation through an Augmented Reality environment. The miniature real robot moves on a horizontal display with real wheels, but it can "kick" the virtual ball through a virtual kicking device.

Table 1.1 briefly summarizes the features of available simulators which are mentioned above. All the robot simulators have similar features, like full rigid-body dynamics, visualization, robot model formats, network support, and are organized by similar architecture as in figure 1.1.

The physics engine plays an important role in robotics simulation. It performs collision detection, resolves collisions and other constraints, and calculates properties (such as position and speed) of all the objects. There are many physics engines available, both commercial and open source. Some of them are high precision engines that require higher computational power while others sacrifice this accuracy to work in real time. Described below are four widely used physics engines:

Open Dynamics Engine (ODE) [69] is an open source, high performance library for simulating rigid body dynamics. It is fully featured, stable, and mature. It is platform independent with an easy to use C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects in virtual reality environments and virtual creatures. It is currently used in many computer games, 3D authoring tools and simulation tools.

Bullet Physics [1] is a professional open source collision detection, rigid body and soft body dynamics library. It supports discrete and continuous collision detection including ray and convex sweep tests. Furthermore, it has be integrated into professional animation software Maya and Blender.

PhysX [15] is a proprietary realtime physics engine developed by NVIDIA Corporation. It supports rigid body dynamics, soft body dynamics, ragdolls and character controllers, vehicle dynamics, volumetric fluid simulation, and cloth simulation including tearing and pressuring cloth. One feature PhysX has that other engines do not have is hardware acceleration. It can be accelerated by either a physics processing unit (PPU) or a CUDA-enabled GeForce graphic processing unit (GPU). PhysX has provided physical simulation for more than 300 games.

IBDS [54] is a free library for dynamic simulation of multi-body systems in C++. It simulates rigid bodies, particles, many different joint types and collisions with friction. It uses a new method for the dynamic simulation of linked rigid body systems where forces are systematically substituted by impulses. In this way the simulation is performed in a very direct way without solving complex systems of differential equations. The basic principle of the method is a new interactive procedure the calculates impulses which conserve joint constraints.

It is also possible to use multiple physics engines within one application. The **Physics Abstraction Layer** (PAL) [7] provides a unified interface to a number of different physics engines. The PAL project provides a set of standard benchmarks allowing developers to directly compare the physics engines and select the engine that provides the best solution in terms of computational efficiency and physical accuracy [8].

Simulation has become an essential tool for developing robot software. In contrast to a real robot, simulation gives developers access to cost prohibitive or unavailable robots. Additionally, the actual working environment maybe not accessible while the simulated environment is always available. Virtual access to different working environments makes virtual testing cost efficient. Simulation is also safer for researchers; and allows them to safely refine their assumptions about the robot and their algorithms. For developers, another important attribute of simulations is repeatability, which is important for debug-

ging. The same scenario can be precisely generated to trigger a known error and checked for a solution. In addition, all vital data can be logged to give developers an understanding of inconsistencies in their algorithms. Therefore, simulations allow robot software development to be more effective, cost effective, and efficient.

However, the available simulators do not meet the ideal development scenario, which was described before. An algorithm that works perfectly in simulation is not guaranteed to work at all under actual environmental conditions. Firstly, there are inconsistencies between simulation models and real world applications. Secondly, for some simulation-system/real-hardware combinations, substantial code and command interface changes must be made. These changes may introduce new bugs. Lastly, it is difficult to isolate failures due to the lack of directly debugging and repeatable trials on the real robot.

1.1.3 RoboCup

The Robot World Cup (RoboCup) initiative is an attempt to foster artificial intelligence and intelligent robotics research by providing a standard problem where a wide range of technologies can be integrated and examined [42]. There are different leagues that make it attractive as an environment for researchers to focus on a set of specific problems on the way to the final goal – "By the year 2050, a team of fully autonomous humanoid robots will be developed. They will be able to win against the human world soccer championship team." The Simulation League and the Standard Platform League both have long histories and big communities. Next, we will describe the current state of these two leagues.

Simulation 3D League The Simulation League has existed from the beginning of RoboCup in 1997. The 2D simulator only runs in an abstract virtual environment. The participants concentrate mainly on coordination and cooperation of robot teams. Because the simplified robot model in a 2D simulator is far from real robots, Kogler [46] introduced a three-dimensional physical simulation – SimSpark in RoboCup 2004, where sphere shaped agents ran around the field. SimSpark increases the realism of the simulated environment and makes it more comparable to the other RoboCup league environment. However, it is still able to simulate more players on the field than other leagues. Thus, the Simulation League rises to higher level of research on larger groups of slightly more realistic soccer playing robots.

In 2007, the humanoid robot (based on HOAP-2 [14] from Fujitsu) was introduced into the 3D Simulation League. The NAO robot from Aldebaran [2] has been used since 2008. This opens up opportunities for research at lower levels of control on humanoid robots as well as higher level behaviors in humanoid soccer. It is getting closer to how humans actually play the game. The participants have to create soccer playing agents for a team which deals with the higher complexity of a human shaped body in the (simulated) physical environment. The hope is that the results can be transferred to humanoid robotics more easily.

As a consequence of the changes to the 3D simulator, we observed a temporary shift of

Figure 1.2: A nine versus nine game was playing in the 3D Simulation League at RoboCup 2011 in Istanbul.

problem solving to the more basic issues of motion control. This is related to restrictions on tactics and how to plan a team play when the basic skills do not allow it, e.g., for a good pass.

At the same time, simulation allows for more players than it does with real robots. In 2011, the field size of the 3D Simulation League increased to 21×14 square meters, and the games are played nine against nine (See Figure 1.2). This means, coordination and cooperation between robots becomes important as in the 2D simulation. There is a two tired problem: developing the skills for better tactics on one hand, and realizing tactics based on the actual skills on the other hand.

Standard Platform League In the Standard Platform League (SPL) all teams use the same robotic platform. This gives scientists the opportunity to concentrate on the software development only. On another hand it provides a better platform for comparing the developed algorithms in the game, since no team may gain advantages by changing the robot hardware (e.g., using stronger motors). Figure 1.3 shows a scene from a game in the Standard Platform League.

The robots in SPL run autonomously as in the Simulation League. There is no particular external control by either humans or computers. Perception is based on sensors (cf. Table 1.2) and calculated by the on-board computer. Communication is possible via wireless LAN, but it may be corrupted by environmental conditions. The on-board computer is also used for planning and motion control. The runtime system provides special middle-ware for communication between software and hardware.

Until 2008 the four-legged Sony Aibo robot [70] was used as the standard platform. Starting in 2008 the humanoid robot NAO produced by Aldebaran, was used. Because

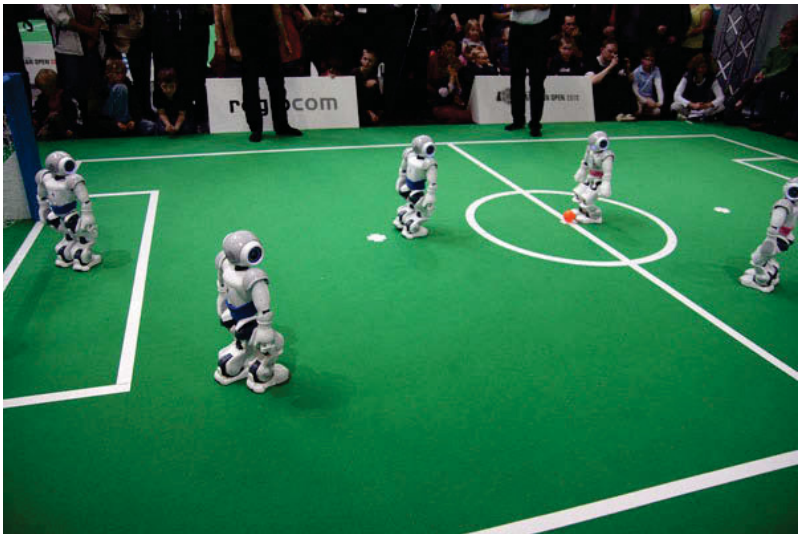


Figure 1.3: A three versus three game played in the Standard Platform League at the German Open 2010 in Magdeburg.

of the 10-year difference, the NAO can benefit from progress in technology. But, the restriction of processing power is still a bottle neck, because reactions have to be in real time. For example, perception and control calculations should be completed during the cycles used by the frequency of images (30 frames per second). Otherwise the robot acts on obsolete data. The runtime system uses a special middle-ware called – NaoQi [2].

Of course, biped motion is much more complicated than quadruped motion, and skills like kicking and dribbling are important. For the Aibo, the RoboCup teams developed a walk of about 50cm/s which was about three times faster than the original walk. This was mainly done using machine learning [22].

Comparison of Two Leagues Since the Standard Platform League and the Simulation 3D League use the same robot, NAO from Aldebaran Robotics (see figure 1.4), it provides a good opportunity of cooperation between the two leagues. However, there are also some difference between them.

Because of different research targets, there are some inconsistencies between simulated NAO in SimSpark and real NAO, especially when it comes to sensors and actuators (See Table 1.2). For example, the simulation league community doesn't want to be limited by current robot hardware technology.

The left hip and the right hip in the real NAO are physically connected by one motor so they cannot be controlled independently. The simulated robot has a motor for each joint. Furthermore, the robot program has to communicate with NaoQi for sensing and acting. The network connection is established between the agent program and the simulator. Last but not least, the real robot has to process images to understand the world. While the simulated robot gets positions of objects via virtual vision system directly.

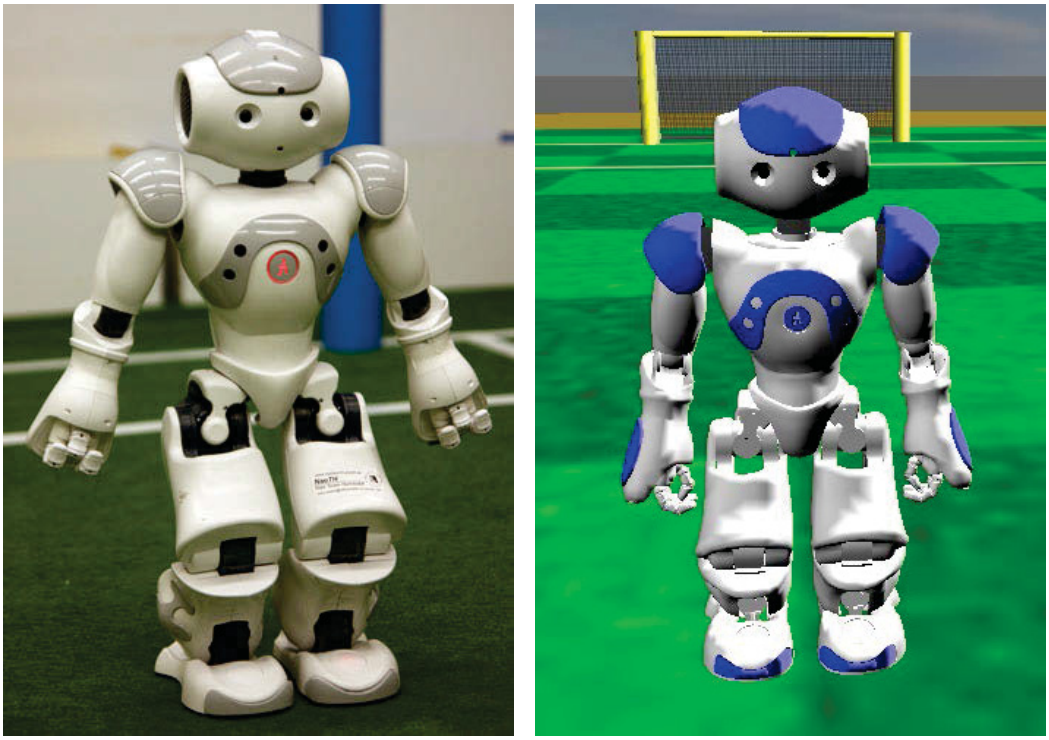


Figure 1.4: The RoboCup edition NAO V3+ robot (left) from Aldebaran Robotics and simulated Nao robot in SimSpark-0.2.2 (right).

Table 1.2: The difference between the real NAO robot and the simulated robot in *SimSpark*.

	NAO V3+	NAO in SimSpark
Degree of freedom	21	22
Joint control	angle, stiffness (100Hz)	velocity (50Hz)
Vision	2 cameras (30FPS)	vision information (16.7FPS)
Accelerometer	3 axises	3 axises
Gyrometer	2 axises	3 axises
Force sensor in each foot	4 force sensitive resistors	6 dimensions force sensor
LEDs	in chest, ears, eyes, feet	no
Loudspeakers	2 in ears	no
Microphones	2	no
Sonars	2 in front of chest	no
IR transmitter/receiver	in eyes	no
Bumper	at tip of each foot	no
Platform interface	NaoQi	network
Team Communication	WLAN	say/hear

Table 1.3: A comparison of games in the Standard Platform League and the 3D Simulation League in RoboCup 2011.

	Standard Platform League	3D Simulation League
Field size	6m×4m	21m×14m
Goal size	1.4m×0.8m	2.1m×0.8m
Number of robots	4	9
Game	without stop	with kick in, goal kick, etc.
Referee	human	computer
Duration	20 min	10 min

Besides the robot, the game in different leagues are organized in different ways (See Table 1.3). First, the field size and the goal width in the Simulation 3D League is bigger than for the Standard Platform League. Each team has nine robots in the Simulation League while only four robots are used in the Standard Platform League. Furthermore, the duration of one Standard Platform League game is twice as long as that of a simulation game. Last but not least, in comparison to the Standard Platform League, the rules of the 3D Simulation are more like that of human soccer rules. A computer is used as referee in simulation unlike a human used in the SPL.

1.2 Motivation and Approach

In every field there is a corresponding gap between theory and practice. This gap first appears during the process of constructing an abstract model. This process involves ignoring some lower-level details while retaining those aspects believed to be important to the goals of the analysis. The gap frequently widens when the initial model proves too difficult to analyze. In this case additional abstractions and assumptions are often made. They are motivated by the goal of improving analytical tractability rather than by the properties of the underlying object of study. Robotic simulation is one of the classic examples. One characteristic of all simulations is that they can only approximate the real world. This inherent deficit is the *Reality Gap* [47]. Obviously, the wider the gap, the more concern there is as to whether the results obtained from simulation carry over to the "real" thing.

1.2.1 Motivation

For most of the mentioned simulators, transferring controllers developed in simulation to real robots is difficult. This is one of the major challenges when using it comes to evolutionary algorithms. There are several examples: Hein [34] reported that the control program developed under simulation fails to behave similarly in reality.

This is because of evolution and the fact that optimization algorithms heavily explore the features of the environment, i.e. certain characteristics of the motors or the properties

of the ground. Hein [34] reported that servo motors, friction, and stiffness of the robot's body are difficult to model. Cominoli [14] recognized the backlash of the servo affected the humanoid robot. In [78–80], Zagal et al. proposed the Back to Reality paradigm which is able to adapt parameters automatically by comparing robot controller behaviors in reality and in simulations. Laue et al. [47] use a multi-staged approach for automatically optimizing a large set of different simulation parameters for the AIBO robot.

Systematic verification and validation of the simulation model was not in sight [58]. As in the fields of simulation, models containing errors can produce arbitrary results. So proving the correctness of a model is a matter of importance. To make sure the validity of the robot models, NIST proposes standardized test methods that can be easily replicated in both computer simulation and physical form [61]. The real robot can be tested, and the computer model can be finely tuned to replicate similar performances on equivalent tests.

At present, there are eight different soccer leagues in RoboCup, which focus on different sub tasks. There is a lot of work being repeated in the different leagues while solutions for the same and similar issues exist in another league. For example, self localization and biped locomotion are common tasks of a soccer robot. It would be useful to achieve and study synergy effects for the same challenges in different leagues.

Robotics research teams who have real robots also develop simulators for their robots. Designing and implementing a good robot simulator is a difficult and time-consuming task. It makes sense to reuse the existing work. As simulation teams want to apply their solutions to real robots, it makes sense to integrate simulation teams and real robot teams.

There are already some collaborations between researchers from different leagues. Dylla et al. [23] investigated soccer model in a way that they are usable for multiple RobCup soccer leagues. Mayer et al. [50] proposed a road map to collaboration between the Soccer Simulation League and the Humanoid League. The USARsim [73] simulator is used as bridging tool between the Rescue Real Robot League and the Rescue Simulation League. The humanoid team Humboldt [34, 35] tried to integrate existing research results from the simulation into the Four-legged League.

Since 2008, the Standard Platform League and the Simulation 3D League use the same robot, NAO, from Aldebaran Robotics. This opens up opportunities to research within both leagues which focuses on different control levels, i.e. the individual skills of the robot is the most important in the Standard Platform League, and the tactics are the main topic in the Simulation League. We propose to develop a joint team of the Simulation 3D League and the Standard Platform League. The team can benefit from both leagues which seamlessly integrate simulation with real robotic hardware. Ideally, it is possible to migrate the results of the simulation 3D team into the real NAO robot without changing code.

Although the Simulation League doesn't want to be limited by the robot hardware, it is impossible to foresee the future of robots with new materials and their actuating and sensing capabilities. If we want to ignore the reality gap and simply deal with artificial worlds, this can be done in any computer game, and then there is no true connection to robotics. Simulation without reference to existing real robots cannot produce useful

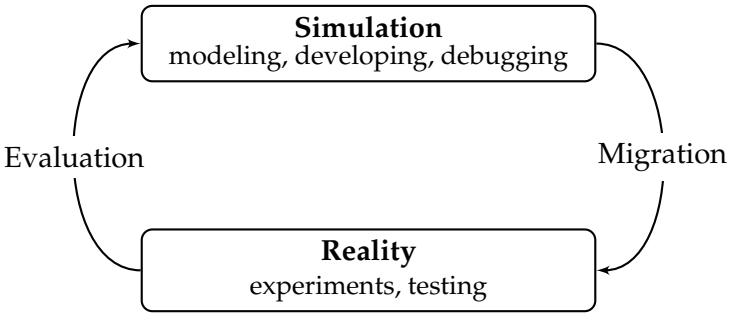


Figure 1.5: Robot development cycle by migrating simulation results to real robot.

results. To deal with the right problems, we need evaluation in reality, i.e. by comparison with real robots. This evaluation is possible only if the simulator simulates existing hardware robots.

We believe that it is possible to develop accurate simulators at a level in which it is possible to learn complex behaviors with the successful transferring of these behaviors to reality. It is important to gain more experience and to find better methods to close the gap between real and simulated physics. On the one side, this can be done by improving the simulator, and on the other side by further development of hybrid techniques which for example, combine learning in simulation and transfer results to real robots. This will help the whole robotics world.

1.2.2 Approaches

The goal of this study was to create a mechanism that provides a smooth gradient to migrate a robot from a purely virtual world to an entirely real implementation. As shown in Figure 1.5, the robotic algorithm development cycle is proposed. We will use more learning algorithms in the simulation and transfer them to a real robot.

The gap between theory and practice can often be narrowed by incrementally removing some of the simplifying assumptions, and studying the effects they have on the earlier analysis. Another gap reduction technique is to perform experimental studies on the models that are mathematically intractable. And, finally, perform experimental analysis of the simulation in ways that can be generalized and adapted to other situations.

This raises questions such as: *How to measure the outcome of simulation? How to improve the accuracy of the current simulator for humanoid robots?* and *How to develop control programs in the simulation and transfer them to real robots?*

The contributions within this work are mainly related to simulation and motion for humanoid robots. They can be summarized as follows:

- A seamless migration of code between real and simulated robots.
- Evaluation of the simulation data results from the robot's sensor and the external sensor.

- Modeling sensors and actuators for humanoid robot simulation, and optimizing parameters by evaluating the results.
- Developing and optimizing motions for humanoid robots in simulation, and transferring them to real robot.

The research was done in our RoboCup team, Nao Team Humboldt, which started in 2008 in the Standard Platform League, and has participated in the 3D Simulation League since 2010. In contrast to other universities where it is usual to set up different teams for different leagues; we use the same source code to play in both leagues. We developed a unified interface for real robot and SimSpark, and use a common core program for both platforms. Consequently, we perform some "gap analysis" as well. In the process we obtain a better understanding of the gaps that exist between simulation and reality, and how they can be addressed.

We offer an overview of the implementation of our team in the next section. For more details about the modules of cognition and motion in our team, please refer to our publications [9–11, 51, 52, 74–76].

NaoTH Framework and Soccer Agent

One research target is to migrate the robotic control software from the simulation to a given robotic hardware. In particular, we aim to use the same program (source code) to participate in both the Standard Platform League and the Simulation 3D Leagues. To achieve this, the architecture strives to seamlessly integrate simulation system with real robotic hardware, and allows simulated and real architectural components to function seamlessly at the same time. The goal is to also be able to integrate existing available infrastructures. The whole project can be divided into three parts: architecture, soccer agent, and tools.

Architecture Humboldt-Universität has a long history with RoboCup. There is a lot of experience and existing code, especially from the GermanTeam [64] with its module based architecture. In order to integrate different platforms, we divided the project into two parts: a platform independent part and a platform specific part. The platform independent part is the Core, which can run in any environment, and all the algorithms are implemented here. The platform specific part contains code applied to the particular platform. In the Core, different modules are implemented under the module based architecture (see Figure 1.6).

At present, our robot control program can not only run in the real NAO robot and the SimSpark simulator, but also in the Webots simulator and the Log simulator. We implemented an abstraction interface in a sense - think - act loop. With the unified platform interface(see Figure 1.7), the Core of our program is independent of the platform which it runs. And all the platform dependent codes are in separate platform implementations.

As mentioned in section 1.1.3, there are some differences between simulation and real robot. Compared to the real NAO robot, some devices are missing in the simulations,

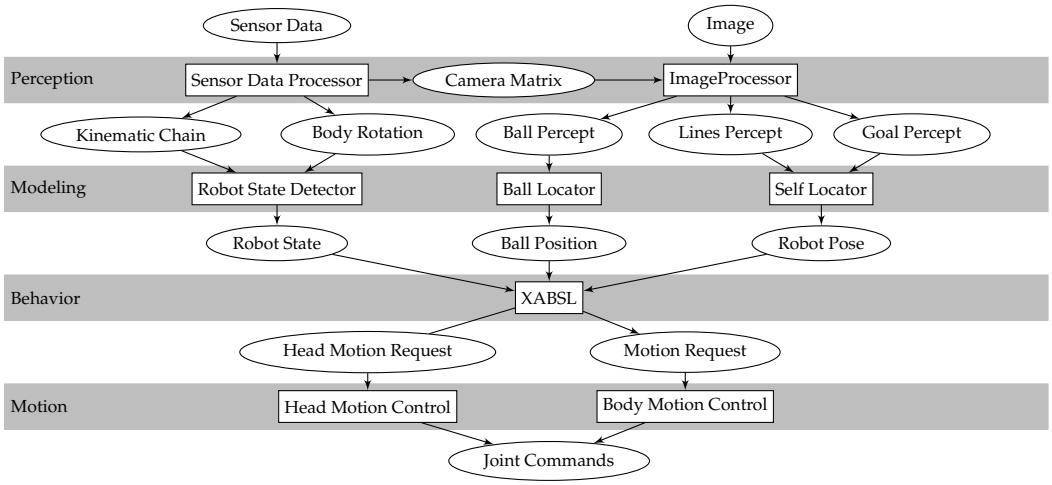


Figure 1.6: The main flow of information processing in the NaoTH robots. Boxes denote modules, ellipses denote the representations that are needed to exchange information between the modules.

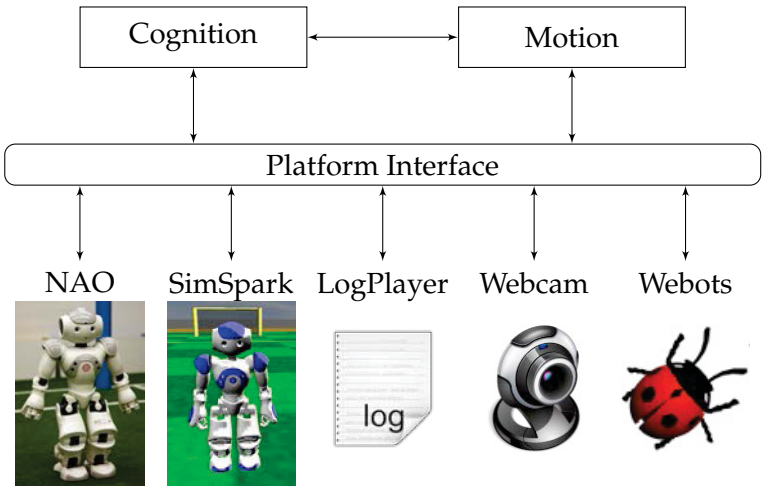


Figure 1.7: The framework of the unified platform interface in NaoTH-2010 and the Core (Cognition and Motion) of the program can run in different platforms with different modules.

such as LEDs and sound speaker. These platforms skip unsupported devices, and use different modules. For example, the camera (image sensor) is not used in the 3D Simulation League. Thus, the controller can disable the image processing module and use the virtual *see* sensor of the SimSpark simulator to provide perceptions. Therefore, the Core can run on different platforms and enable different modules for different sensors and actuators.

Soccer Agent We divided the soccer agent into two main components due to the process speed requirements: motion process (100 Hz) runs faster than cognition process (about 30 Hz). But the motion and cognition are exactly the same for all the different platforms. The processing is briefly described below.

Motion: We implemented motions using keyframe techniques and inverse kinematics with motion planning. Keyframe nets were developed through teaching and hand coding with the help of our motion editor. We adjusted key frames for the different platforms. For parametrized dynamic motions such as omnidirectional walks and kicks, we investigated inverse kinematics with the sensor feedback [51, 75] and evolutionary approaches to optimize the parameters.

Cognition: The camera matrix is computed by the kinematic chain. A Kalman filter was applied to generate a local ball model. Global ball positions are communicated between players to coordinate team behavior. Percepts of landmarks in vision and motion status are passed into a particle filter to calculate the global position of robot.

Behavior: It is executed through a hierarchical state machine - known as the Extensible Agent Behavior Specification Language (XABSL) [38]. The behavior code generation is supported by the related editor, debugging, and visualization tools. It allows layered usage of different behaviors - from low-level motions to high-level cooperative team skills. It is easy to describe some behavior from soccer theory. For example, Figure 1.8 describes the role change between four basic states.

The decision to change between states depends heavily on the characteristics of the robot and the environment, i.e. the conditions for decisions are different in different leagues. We chose some parameters particularly for different leagues. The better alternative is to create some symbols by lower level control modules which makes the decision processes more general. For example, the motion module provides parameters of motion skills like the walking speed. Then the behavior module can base its decisions on real values. We have successfully used related behaviors to play in both leagues, i.e. for skills like “go to ball and kick”, and “dribble”, respectively.

Tools We developed a debugging system which can be switched on or off during runtime. Debug results are transferred over the network and monitored/visualized using RobotControl. It is a robot data monitoring and debugging tool implemented in Java to be used on arbitrary computer systems (see Figure 1.9). But it is limited by connecting to one robot only. In order to do the analysis and develop the team behavior of a robot soccer team, we need to connect to all the robots at the same time. A new tool – TeamControl has been developed.

1.3 Summary

Simulation is an effective and practical method to study and explore the real world problems. The ideal robot development scenario is developing algorithms in the simulation first, and then moving them to real robots without any necessary changes. However,

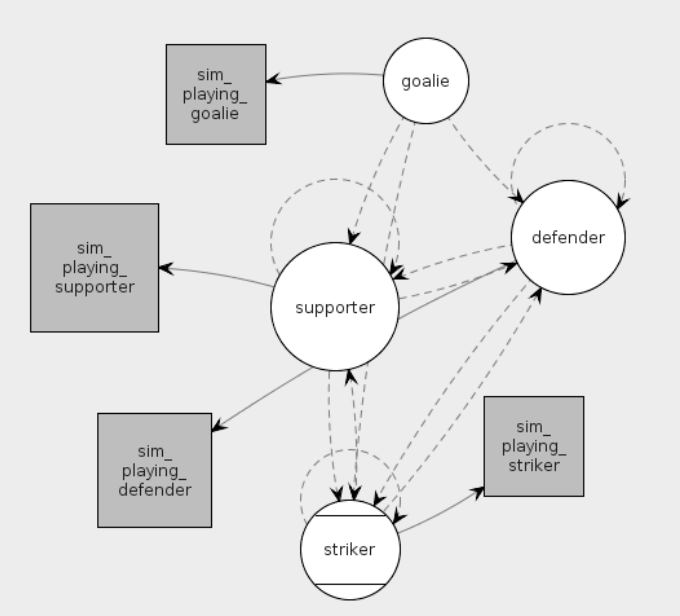


Figure 1.8: The role changing behavior described by XABSL. The circles are states and gray boxes are sub-options. This example shows that the robot changes its role between goalie, defender, supporter and striker, and calls different sub-options in different states.

there were no successful examples of generating complex behaviors for humanoid robots in a simulation and transferring them to reality due to the reality gap.

Since simulation can offer valuable results for robotics only in close connection to real robots, we investigated creating an approach that provides a smooth gradient to migrate a system from a purely virtual world to an entirely real implementation in this thesis. First, we evaluated the performance of the simulator and found out the inconsistencies between simulation and reality. Then, we improved the realism of simulator by building better models and a parameter optimization. Finally, we developed motions for humanoid robots in simulation, with the particular help of evolutionary algorithms and machine learning. We transferred and tested the resulting motions to real robots. We developed a particular RoboCup team to play both in the Standard Platform League and the Simulation 3D League. As a result, we illustrate a meeting point between the communities of simulation and real robot.

The work is structured as follows: In chapter 2 methods of simulation evaluation are presented, and the SimSpark is evaluated. In chapter 3, these evaluation methods and results are used to create realistic simulation for humanoid robots. Chapter 4 develops motions for humanoid robot in simulation, and transfers them to real robots. The work is summarized in chapter 5.

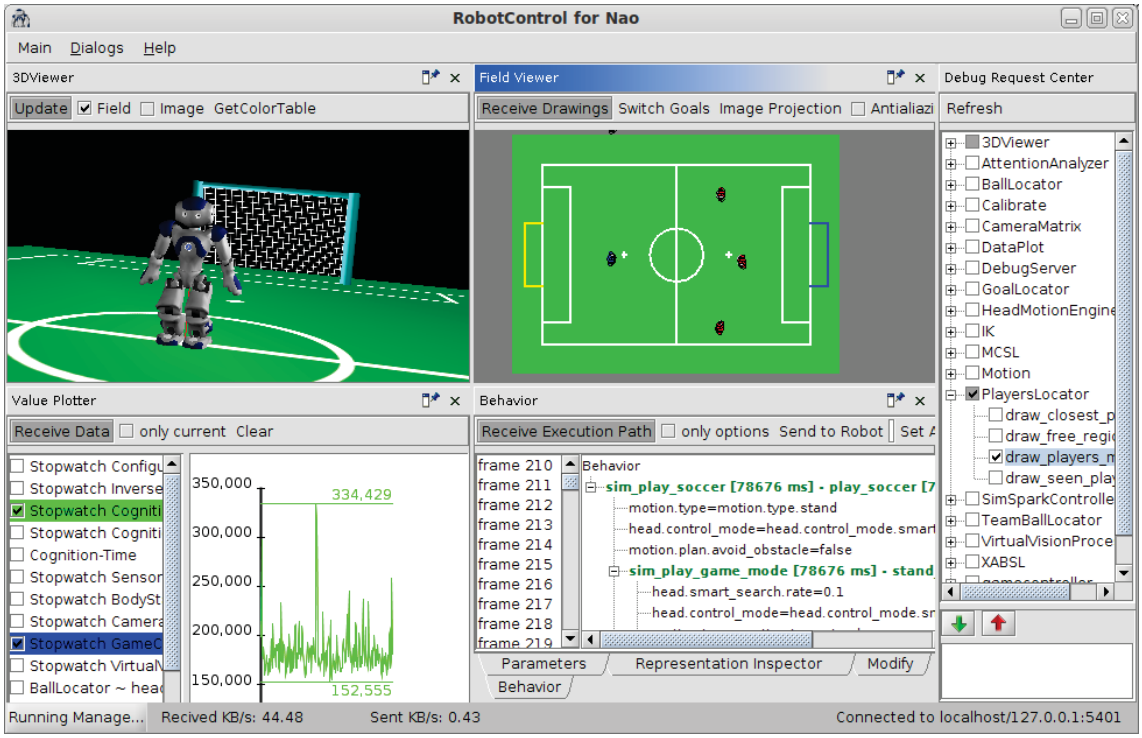


Figure 1.9: The RobotControl program contains different dialogs. In this figure, the left top dialog is the 3D viewer, which is used to visualize the current state of the robot. The left bottom dialog plots some data. The middle top dialog draws the field view. The middle bottom shows the behavior tree. And the right dialog is the debug request dialog which can enable/disable debug functionalities.

2 Robotics Simulation and Evaluation

> In order to be scientific, a theory must be falsifiable.
>
> — Popper, K.R. [62]

Robotics simulation makes the development easier. However it requires proper models that behave closely to real ones. This is especially true when working with humanoid robots which have a high number of degrees of freedom. Otherwise, the simulated robot might not only behave unrealistic but also fail completely.

Unfortunately perfect representation is never expected, since the model is an abstraction of a system. In [68], Shannon pointed out that "it is not at all certain that it is ever theoretically possible to establish if we have an absolutely valid model; even if we could, few managers would be willing to pay the price."

Therefore, the model of robot in simulation should be proper, i.e. accurate and affordable. If developers simulate a robot with models which are not accurate enough, then their results can be meaningless. In order for simulation results to be valid for implementation on real robot, the accuracy of the simulation model must be verified. To ensure the validity of the robot models, NIST [61] proposes standardized test methods that can be easily replicated in both computer simulation and physical form. Understanding and applying particular principles and employing proper testing techniques throughout the life cycle of a simulation study are key factors in increasing the probability of success.

In this chapter, we evaluate the simulation at the model level and at the final results level. The work is organized as follows: the methods to evaluate simulation are introduced in section 2.1. In section 2.2, we evaluated the conceptual model of the SimSpark by conducting experiments. In section 2.3, we evaluated the simulation results of both static motion and dynamic motion. In section 2.4 at the end, is a short summary and discussion.

2.1 Simulation Evaluation

Simulation is used in a lot of industries, such as aerospace, nuclear, and robotics. There are some frequently asked questions, for example: Did we build the simulator correctly? Is its fidelity appropriate? Have the model and simulation been built so that they fully satisfy the required specifications?

For a given simulator, we can answer these questions in two categories: design and implementation. Some researchers [4, 58] proposed two phases of evaluation: verification and validation. The purpose of verification is to check whether the implementable

model reflects the conceptual model. The aim of verification is that the computer code has no programming errors (bugs) left. Are we developing the 'right' simulation for the purposes we have determined? the validation should answer this question [4]. Validation determines whether the simulation is an accurate representation of the system under study. The validation measures the difference between simulation outputs and the real world. Then it is possible to decide if the difference is small enough to the simulation in a specific application and whether the result will have the required and expected accuracy.

Two types of errors may occur while conducting a simulation study. *Type I error* occurs when the simulation results are rejected, but in fact they are sufficiently credible. *Type II error* occurs when invalid simulation result is accepted as if it is sufficiently valid. To ensure the validity of the robot models, some test methods that can be easily replicated in both computer simulation and on real robot are necessary.

However, there are so many options involved in simulation. In order to improve the simulation, the validation must show which parts of the simulation are not appropriate. Therefore, validation must be addressed at two levels:

Model Validation is the determination that the assumptions underlying the proposed conceptual model are correct and that the proposed simulation design elements and structure (ie the simulation's functions, their interactions, and outputs) will lead to results realistic enough to meet the requirements of the application.

Results Validation compares the responses of the simulation with known or expected behaviour from the subject it represents to ascertain that those responses are sufficiently accurate for the range of intended uses of the simulation.

The possible outputs of simulation are extremely large. It is impossible to check all outputs in practice. Hence, a simulation cannot be completely 100 percent validated for all practical purposes. The answer to the question, Is this simulation validated? should always be: Yes, for the conditions specified in the validation report. Therefore, validation is typically performed on those aspects of a simulation that are known to be important in a particular application.

In both *model validation* and *result validation*, the difference between a good validation result and a bad validation result can not be determined without a good definition of what it is being validated against. For example, will the simulation be validated against analytical data, measured data, or another simulation? Each of these real worlds has inherent drawbacks and limitations that can make or break the apparent validity of a simulation.

The validity of a model is always partially dependent upon the desired applications. This is clearly illustrated by the validation criteria listed below (see also [43, 60]):

Empirical validity correspondence between measurements and simulations;

Theoretical validity consistency of a model with accepted theories;

Pragmatic validity capability of the model to fulfil the desired purpose;

Heuristic validity potential for testing hypotheses, for the explanation of phenomenon and for the discovery of relationships.

The simplest form of evaluation is comparing the result from a simulation with observations from the real world and making a judgment about whether the result is accurate enough for its application to the problem. To achieve this it is necessary to take measurements on real systems. Obtaining real world data can be scarce or abundant. Different techniques must be used for different systems. Evaluation techniques used in this thesis are discussed next.

Evaluation based on measured data Comparing the simulation result to measured data is the simplest method and the most convincing evaluation. It is always preferred when the data in the real world can be measured.

In the evaluation of robotics simulation, we have to compare a time series of simulation model output with a historical time series of real output. The visual comparison can provide an initial clue about the difference between simulation results and measured data. The common method used is plotting data such that the horizontal axis denotes time and the vertical axis denotes the real and simulated values respectively. The resulting figure can reflect the accuracy of the simulation and the phenomena of interest.

Another simple technique is the Schruben-Turing test whereby the analysts present a mixture of simulated and real time series to their clients, and challenge them to identify the data that were generated by computer.

Furthermore a whole range of measured variables can be used to check the correspondence between measured data and simulation results. We can define quantitative function for the time-discrete case, which represents a degree of correspondence between the measured response z_i and the result of the simulation y_i. The following formula shows the first possibility:

$$Q_1 = \frac{1}{n}\sum_{i=1}^{n}(y_i - z_i)^2 \tag{2.1}$$

where n denotes the number of data samples.

Another possibility is defining a normalised degree of inequality:

$$Q_2 = \sqrt{\sum_{i=1}^{n}(y_i - z_i)^2} \Bigg/ \left(\sqrt{\sum_{i=1}^{n} y_i^2} + \sqrt{\sum_{i=1}^{n} z_i^2}\right) \tag{2.2}$$

The values of Q_2 lie between zero and one; the values close to one indicating a high level of inequality and the values close to zero indicating a high level of equality between measurement and simulation.

A further approach is recreated in the target functions of simulated annealing and genetic algorithms:

$$Q_3 = \frac{1}{1 + \frac{1}{n}\sum_{i=1}^{n}(y_i - z_i)^2} \tag{2.3}$$

In this case, values close to one indicate a good correspondence and lower values indicate a correspondingly poorer agreement.

Evaluation based on system identification System identification determines the best model of the system within the chosen model's structure. It can construct mathematical models of dynamic systems from measured input-output data. This data-driven approach helps us describe systems that are not easily modeled from first principles or specifications. It can also be used to determine the parameters of a model by minimizing the difference between measured and simulated data.

The results of simulation strongly depend on the parameters of the model. Therefore the procedure of parameter estimation can be used to validate a predetermined model structure. In other words, if measurement errors are negligible, a high standard deviation of the estimated parameters in the identification for different sets of measured data can indicate that the structure of the model does not correctly reflect reality [60].

Evaluation based on the inverse model The inverse models are used for controlling the system, and the inverse techniques for dynamic simulation models can be used for evaluation. In this case the desired output are predetermined. An inverse model in the form of an ideal controller calculates the necessary control. Thus the validity of the model is demonstrated on the basis of outputs supplied from the inputs generated using the inverse model. The criteria of direct evaluation based on measured data can be applied here again.

This method has been successfully applied in aircraft flight simulation in [56]. The desired flight movements were predetermined; and an inverse model in the form of an ideal pilot calculated the necessary control of the helicopter. The validity of the helicopter model is demonstrated on the basis of outputs supplied from the inputs generated using the inverse model. The criteria from direct evaluation based upon measured data, for example, equations (2.1) to (2.3), can again be applied here.

Evaluation based on model hierarchy If a model contains different components and the interconnection of the components occurs directly within the model, then the evaluation of this model can be based upon the evaluation of its components. This approach is similar to the modular architecture in software engineering whereby a system is broken down into component modules, in which each individually functions.

An example of this is the evaluation of the model of a robot, which is constructed from a list of components such as motors, sensors etc. If the actual connection structure of the robot is represented by a model, then the evaluation of the robot model is transformed into the evaluation of the component model.

In this thesis, we prefer evaluation based on measured data if the data can be measured. Data from robot's sensors and external sensor, e.g. Kinect are used for evaluation. When the data can not be easily measured, we use evaluation based on system identification.

For example, we modeled the motor from specifications, and evaluated the model by the procedure of parameter identification. We use our motion engine as the inverse model for evaluation, it is done by comparing the results of the same motion engine in real robot and the simulation. The idea of evaluation based on model hierarchy is also applied here, for example, the motors and sensors are evaluated first, then the whole robot model is evaluated.

Although we cannot say that a model is valid on the basis of simulation experiments, we can say at best that the model is not valid if false results are made. If the simulated model is valid, it follows that the results of the simulation are correct in relation to reality. However, the reverse is not necessarily true! It is possible for faulty models to produce correct predictions by strange coincidence. In principle a greater number of simulation experiments does not change the outcome. The probability that the model is valid increases with the number of experiments. There can never be a rigorous evaluation of a scientific theory. The best that we can do is to develop empirical tests for the theory — fair tests, but the stricter the better — and hold onto the theory only as long as it passed all the tests [62].

2.2 Model Evaluation

We have to evaluate a simulator before we can improve it. As described in Figure 1.1, a modern robot simulator contains physical engine and robot models, which are both built according to models. The current implementation of the SimSpark simulator uses the Open Dynamics Engine (ODE) [69] for physically realistic dynamics simulations. Real robot devices, such as sensors, actuators, mechanics, etc., must also be modeled.

Therefore, the models of robotics simulation can be categorized into two groups: *physical dynamics models* and *robot hardware models*. In this section, we discuss both physical motion models and robot hardware models. We evaluate models of the SimSpark by using methods discussed in section 2.1.

2.2.1 Physical dynamics model

Since robots are made by non-deformable materials (such as metal or plastic), they are usually modeled as rigid bodies in physics. The study in physics of the motion of rigid bodies is *rigid body dynamics*.

Different methods have been proposed to model rigid body dynamics. There are several classes of methods to model contact and collision in rigid body simulation, for example: penalty methods [20], impulses based model [54], and explicit time-stepping formulations [3].

In this section, we discuss basic concepts and algorithms of physics simulation, and present a qualitative evaluation for ODE.

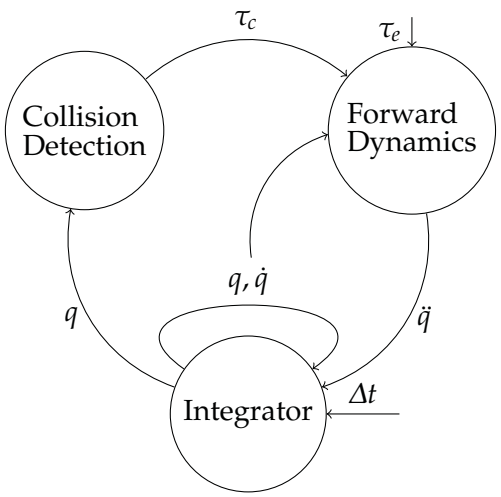


Figure 2.1: The main components of a rigid body dynamics engine. q, $\dot{q}$, and $\ddot{q}$ are position, velocity, and acceleration respectively; τ_c is the force of collision and friction; τ_e is external forces; and Δt is step time of integrator.

Rigid Body Dynamics

A *rigid body* is an idealization of a solid body of finite size in which deformation is negligible and therefore ignored. They occupy space and have geometrical properties, such as a center of mass, moments of inertia, etc., that characterize motion in 6 degrees of freedom (translation in three directions plus rotation in three directions). Even though such an object cannot physically exist due to relativity, robots can normally be assumed to be perfectly rigid. As such, rigid body dynamics is used heavily in analysis and computer simulations of physical systems and machinery where rotational motion is important, but material deformation does not have a significant effect on the motion of the system.

In the dynamic simulation, a computer calculates the forces, accelerations, etc. associated with the motion of a rigid body approximating a physical system. The relationship between the forces acting on the system and the accelerations they produce are modeled by equations of motion (see Figure 2.1). Therefore, evaluating physic models in robot simulation is evaluating these equations of motion.

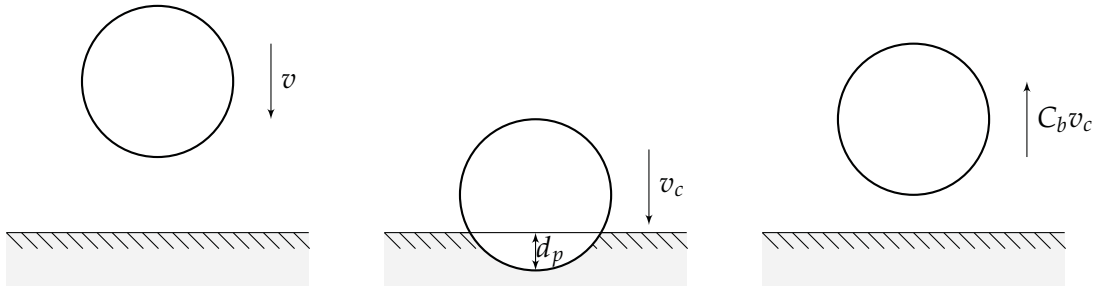
The equation of motion for a rigid body usually is written as follows:

$$\boldsymbol{\tau} = \boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) \tag{2.4}$$

where, $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$ and $\ddot{\boldsymbol{q}}$ are vectors of position, velocity and acceleration variables respectively, $\boldsymbol{\tau}$ is a vector of forces. $\boldsymbol{H}$ is the inertia matrix, it depends on $\boldsymbol{q}$, $\boldsymbol{C}$ is the bias force which depends on both $\boldsymbol{q}$ and $\dot{\boldsymbol{q}}$, it accounts for the Coriolis and centrifugal forces, gravity, and any other forces acting on the object.

In rigid body simulation, two particular calculations are used:

forward dynamics: calculating the acceleration response of a given rigid body system to



(a) before collision (b) collision detected (c) after collision

Figure 2.2: Illustration of the posteriori collision detection: v_c is the speed when collision is detected, d_p is penetration depth, and C_b is the coefficient of the bounce.

a given applied force; and

inverse dynamics: calculating the force that must be applied to a given rigid body system to produce a given acceleration response.

We can express these calculations as follows:

$$\ddot{\boldsymbol{q}} = \mathrm{FD}(model, \boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{\tau}) \tag{2.5}$$

and

$$\boldsymbol{\tau} = \mathrm{ID}(model, \boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}) \tag{2.6}$$

where FD and ID are functions denoting the froward dynamics and inverse dynamics calculations, respectively.

The main task of a physics engine is to solve the forward dynamic problem. For example: What is the motion of the system with given the forces acting on it? The constraints of a forward dynamic system is a relationship that is enforced between two bodies; so that they can only have certain positions and orientations relative to each other.

Collision detection and response in simulation is very important. A small error in any calculation will cause drastic differences in the final results. Physical simulators usually detect collision in one of two ways: a posteriori (after the collision occurs) or a priori (before the collision occurs). The posteriori way is easy to implement (see Figure 2.2).

For the performance, the simulators firstly detects collision at the coarse level, it finds pairs of objects which might potentially intersect. The Axis-aligned Bounding Boxes (AABB) test is widely used in this phase. The expensive collision detection is applied to each pair of potentially colliding objects. This is done with specialized code for each pair of geometry types in ODE. If collision is detected, movement constraints are created between the bodies.

On comparing equation (2.5) with equation (2.4), it is clear that FD must calculate $\boldsymbol{H}^{-1}(\boldsymbol{\tau} - \boldsymbol{C})$. However, the algorithms that implement it don't need to calculate $\boldsymbol{C}$ or $\boldsymbol{H}^{-1}$. The equations of motion for a rigid body system are obtained as the end result of a sequence of mathematical operations. Two main operations that we performed on them

are: 1) collecting equations together to form the equation of a bigger subsystem, and 2) applying additional motion constraints.

Suppose we have a set of N rigid body subsystems; and subsystem i has the equation of motion $\boldsymbol{H}_i\ddot{\boldsymbol{q}}_i + \boldsymbol{C}_i = \boldsymbol{\tau}_i$. If we wish to treat them as a single system, then the equation of motion for that system is:

$$\begin{bmatrix} \boldsymbol{H}_1 & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \boldsymbol{H}_2 & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \boldsymbol{H}_N \end{bmatrix} \begin{bmatrix} \ddot{\boldsymbol{q}}_1 \\ \ddot{\boldsymbol{q}}_2 \\ \vdots \\ \ddot{\boldsymbol{q}}_N \end{bmatrix} + \begin{bmatrix} \boldsymbol{C}_1 \\ \boldsymbol{C}_2 \\ \vdots \\ \boldsymbol{C}_N \end{bmatrix} = \begin{bmatrix} \boldsymbol{\tau}_1 \\ \boldsymbol{\tau}_2 \\ \vdots \\ \boldsymbol{\tau}_N \end{bmatrix} \tag{2.7}$$

Motion constraints are algebraic constraints on the motion variables of a system. They are caused by constraint forces. The equation of motion for the constrained system is

$$\boldsymbol{H}\ddot{\boldsymbol{q}} + \boldsymbol{C} = \boldsymbol{\tau} + \boldsymbol{\tau}_c \tag{2.8}$$

where $\boldsymbol{\tau}_c$ is the constraint force. This force is unknown, but it has one important property that allows us either to calculate its value, or to eliminate it from the equation, as desired: The constraint force delivers zero power along every direction of velocity freedom that is compatible with the motion constraints [24]. This statement says that $\boldsymbol{\tau}_c$ must satisfy

$$\boldsymbol{\tau}_c \cdot \dot{\boldsymbol{q}} = \mathbf{0} \tag{2.9}$$

Three algorithms can be used to solve the equations: the Recursive Newton-Euler Algorithm, the Composite-Rigid-Body Algorithm, and the Articulated-Body Algorithm [24].

Considering the produce of the dynamic simulation described above, a number of factors that influence the overall performance of a physics engine have to be evaluated, including

- Simulator paradigm: constraint based methods, impulse based methods, or penalty based methods.
- The integrator: determines the numerical accuracy of the simulation.
- Collision detection: efficiency and accuracy of collisions in the simulation
- Material properties: friction, hardness, and bounce.

Physics Engine Evaluation

In this subsection, we focus on the performance of ODE for the humanoid robot simulation, especially when it comes to accuracy. We evaluate different aspects of the physics engine through seven experiments.

Integrator Performance The process of simulating a rigid body system through time is called integration. Each integration step advances the current time by a given step size,

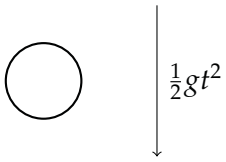


Figure 2.3: Test the performance of numeric integrator with a "free fall" experiment.

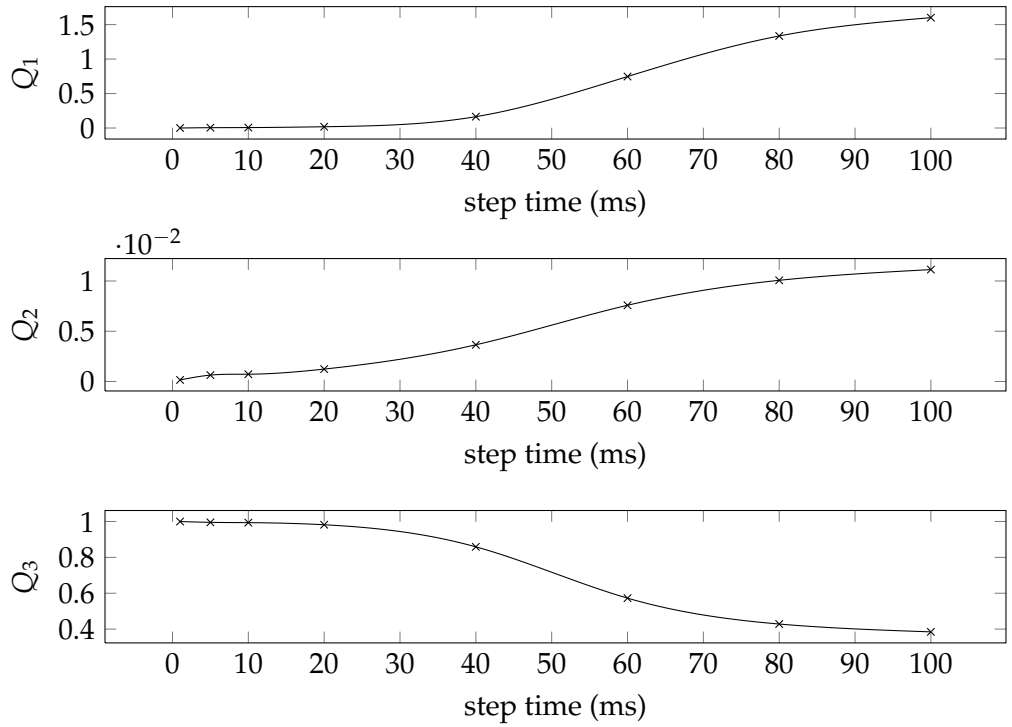


Figure 2.4: Error from cumulative numerical integrator, Q_1,Q_2, and Q_3 are defined in equations (2.1) to (2.3) respectively.

and adjusts the state of all the rigid bodies for the new time value. The performance of the integrator affects the accuracy of the simulation. ODE uses the Euler integrator and fixed time stepping.

We tested the accuracy of the integrator with a "free fall" experiment (see Figure 2.3). A sphere is constructed and allowed to drop using gravitational force. In classical physics, the position of an object without initial velocity can be calculated from *Newton's First Law*. We recorded the positions presented by the physics engine, and compared them with analytical values.

The test was repeated several times with different time steps. The correspondence variables are calculated and plotted in Figure 2.4. It is clear that smaller time step achieves better results, but also takes more computation power. Figure 2.4 shows ODE is not particularly accurate unless the step size is smaller than 20 ms.

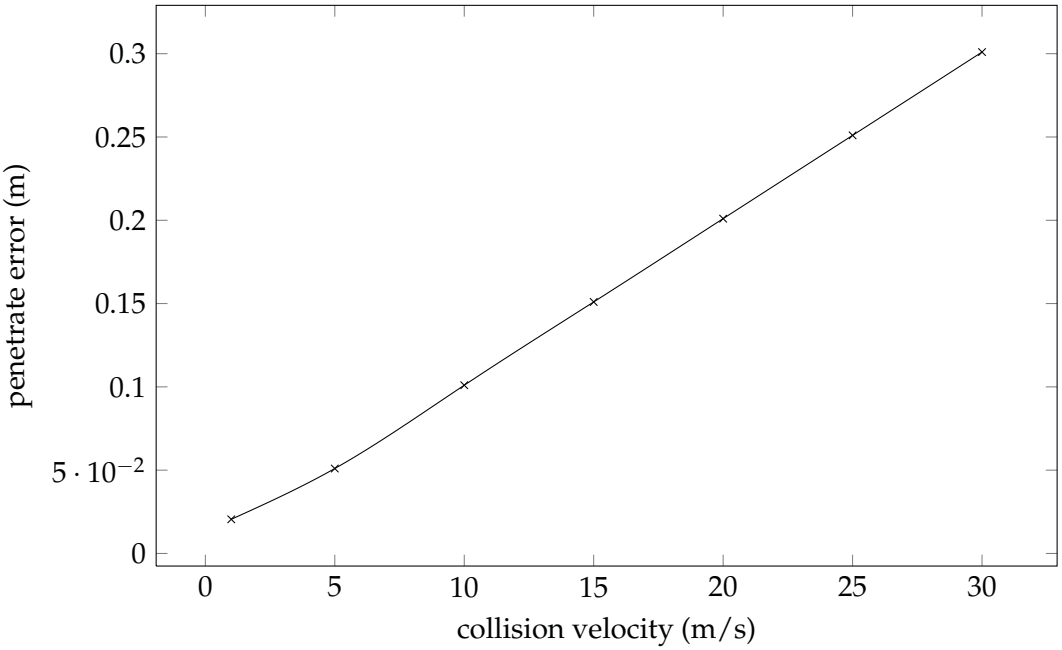


Figure 2.5: Penetration depth for different collision speed, the simulation time step is 10 ms.

Collision Detection The collision system is an essential part of the physics engine, since failure to detect a collision during a simulation leads to incorrect results. The ODE's built-in collision detection engine can handle complex geometries (for example, tri-mesh and height field) as well as simple geometries (for example, sphere and box). Furthermore, an alternative collision detection system can be used as long as it can supply the right kinds of contact information.

A test was created for evaluating ODE's build-in collision detection system. We use a ball (with 1 m radius) to collide with the ground at different speeds. We recorded the penetration between the ball and the ground. Figure 2.5 shows the results: the penetration depth grows while the collision speed increases. This is because ODE only detects collisions after there is already penetration. Therefore the collision could go without being detected, resulting in an object which passes through another, if the relative velocity between two objects is fast enough.

In order to show the effects of step time, we repeated the experiments with the same collision speed but with different time steps. The results in Figure 2.6 confirm that the smaller simulation step time produces the better simulation results. Choosing simulation step time has to take into consideration the speed and size of the objects.

After the collision is detected between two objects, the collision response should be calculated. This is done by adding contact points in ODE, then the friction and bouncing force is calculated by the contact. Both accurate friction and restitution models are critical for robotic simulation.

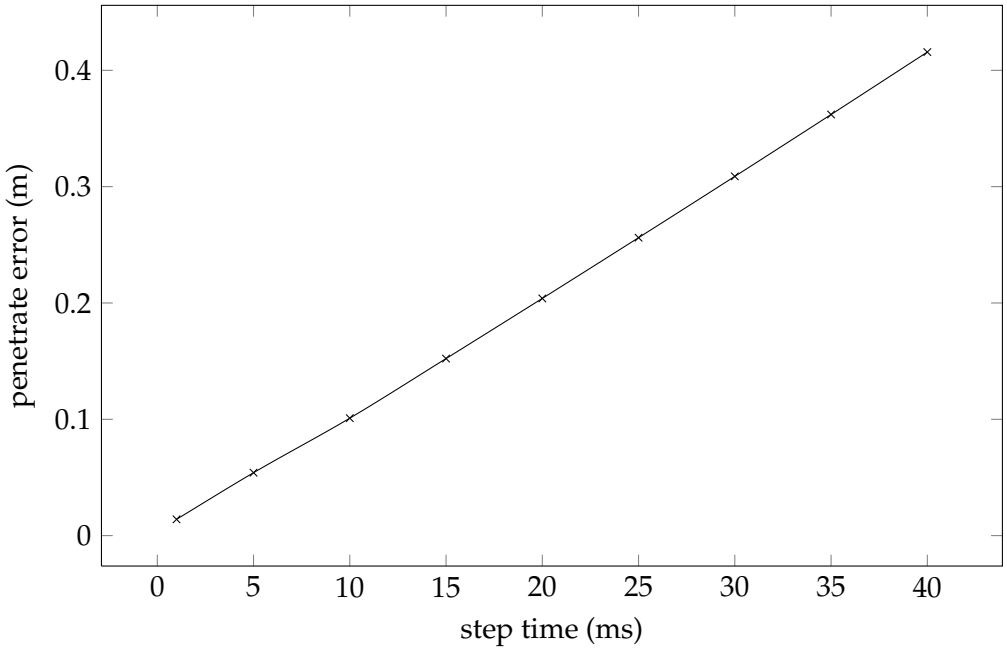


Figure 2.6: Penetration depth for different simulation time steps. The collision speed is fixed at 10 m/s.

It is important to note that representing contacts with contact points is only an approximation. Contact between two bodies is complex in reality. Contact "patches" or "surfaces" are more physically accurate, but calculating them in high speed simulation software is a challenge. Each extra contact point added to the simulation will slow it down more, so sometimes it is necessary to ignore contact points in the interests of speed. For example, when two boxes collide many contact points may be needed to properly represent the geometry of the situation, but we can choose to keep only the best three.

We evaluated the bounce and friction models of ODE in next sections.

Bounce The restitution model is tested by colliding a ball or a cube with the ground. We tested different coefficients of bounce. A graph of a bouncing ball's height over time for different coefficients is depicted in Figure 2.7. The simulated results are close to the analytical values. The small error is mainly caused by collision detection.

The relationship between dropped height and the coefficient of bounce is given in classical physics by:

$$C_b = \sqrt{\frac{h}{H}} \tag{2.10}$$

where C_b is the coefficient of bounce, 0 means the surfaces are not bouncy at all, 1 is maximum bounciness; h is the bounce height; and H is the drop height.

As shown in Figure 2.8, ODE gives good results in simulating the coefficient of bounce,

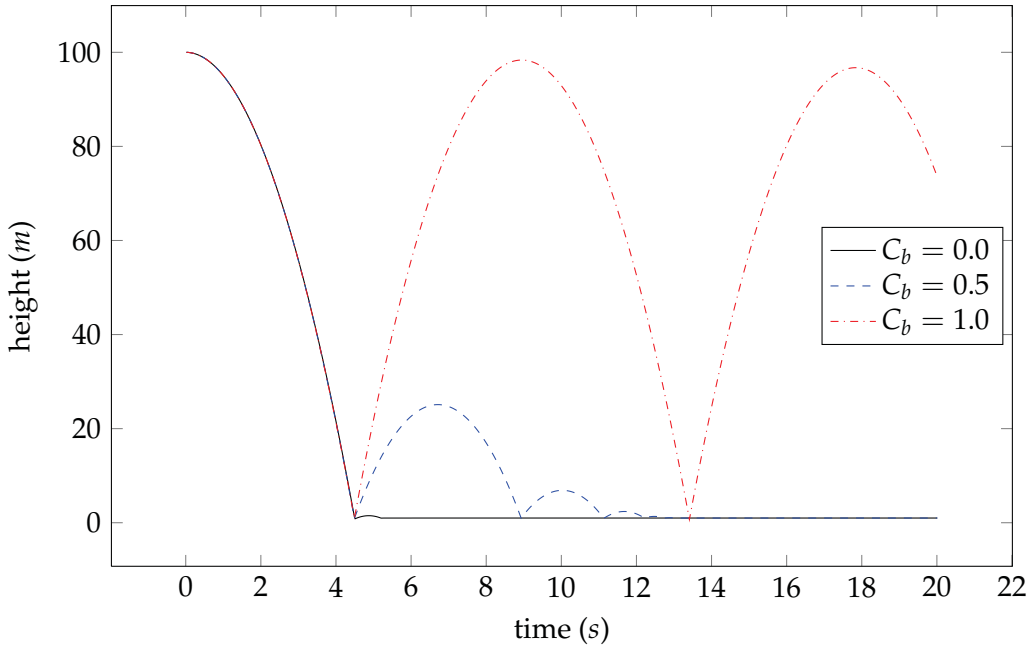


Figure 2.7: A bouncing ball's height over time for different bounce coefficients.

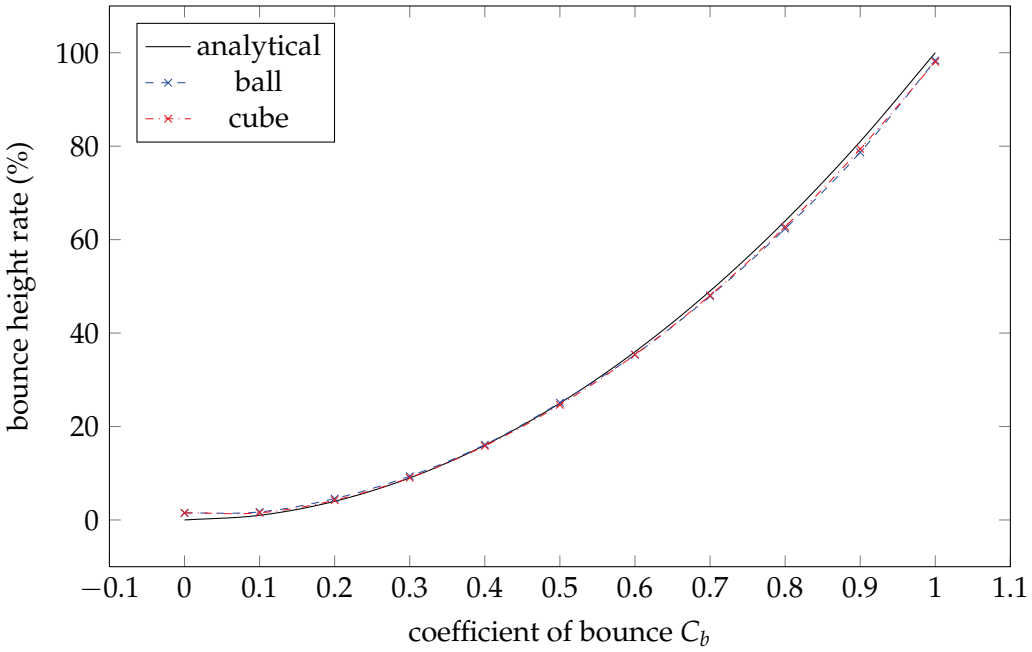


Figure 2.8: The relationship between bounce height and the coefficient of bounce in ODE simulation and analytical calculation.

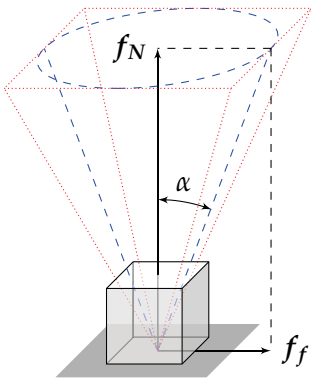


Figure 2.9: Friction model: the (blue dashes) cone is the "friction cone" defined by equation (2.11), where $\alpha = \arctan(\mu)$; and the (red dotted line) pyramid is the "friction pyramid" used by ODE to approximate the friction cone.

i.e. the simulation results match equation (2.10) well. The values of equations (2.1) to (2.3) are $Q_1 = 0.0013$, $Q_2 = 0.012$, and $Q_3 = 0.9999$.

Friction The relationship between the normal and tangential forces present at a contact point is defined as the Coulomb friction model in classic physics:

$$|f_T| \leq \mu |f_N| \tag{2.11}$$

where f_N and f_T are the normal and tangential force vectors respectively, and μ is the friction coefficient.

In ODE, friction models are approximations of the Coulomb friction model for reasons of efficiency. There are two forms: the simplest form is that the maximum friction force present at contact is given as the parameter, the user has to set the value at every step, otherwise this is rather non physical, because it is independent of the normal force. In another form, the friction cone is approximated by a friction pyramid aligned with the first and second friction directions (see Figure 2.9).

We evaluated the friction model by putting a box on the ground, and applying force to it. The relationship between minimum force, which can make the box slide, and friction coefficient μ is show in Figure 2.10. The result is good when the external force and the friction direction in model are in the same direction. The results are less optimal when force and friction are not in the same direction. Unfortunately the later is the most common scenario in simulation. Furthermore, the static friction and sliding friction are not modelled separately, this also does not reflect reality.

Contacts ODE implements contacts as joints. When a joint attaches two parts of body, those parts are required to have certain positions and orientations relative to each other. Most constraints are by nature "hard", this means that the conditions of the constraints are never violated. For example, the two parts of the hinge must always be lined up.

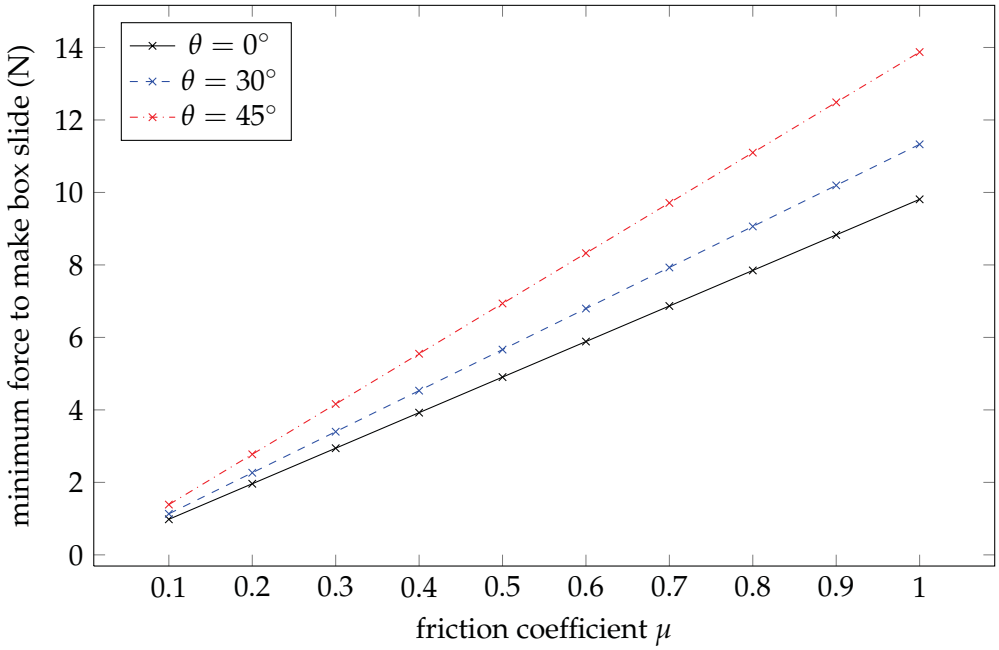


Figure 2.10: The relationship between minimum force which can make the box slide and the friction coefficient μ in the friction model evaluation, where θ is the angle between force and friction direction 1 in the friction model.

However, in practice constraints can be violated by unintentional introduction of errors into the system. Conversely, there are some "soft" contacts which are designed to be violated. For example, the contact point that prevents colliding objects from penetrating is hard, so it acts as though the colliding surfaces are made of steel. But it can be made into a soft contact to simulate softer materials, thereby allowing some natural penetration of the two objects when they are forced together. In such cases, like Figure 2.11, the force of the contact points can generally be modeled by the spring and damper system:

$$f_c = -k\boldsymbol{e} - c\dot{\boldsymbol{e}} \tag{2.12}$$

where $\boldsymbol{e}$ is the error of contact, k and c are spring constant and damping constant respectively.

There are two parameters that control the distinction between hard and soft contacts. The first one is the Error Reduction Parameter (ERP) which controls the force applied to the joint to brings it back into correct alignment during each simulation step. The second is the Constraint Force Mixing (CFM) value which can take the system away from any singularity in mathematical calculation. What is actually happening is that the contact is allowed to be violated by an amount proportional to CFM times the restoring force that is needed to enforce the contacts.

In other words, ERP specifies what proportion of the joint error will be fixed during

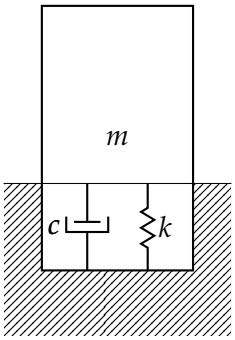


Figure 2.11: The spring and damper representation of contacts.

the next simulation step, and the softness will increase as CFM increases. We can achieve different effects by adjusting the values of ERP and CFM. In fact, the spring and damper system can be simulated by selecting ERP and CFM; the corresponding ODE parameters can be determined by:

$$erp = \frac{k\Delta t}{k\Delta t + c} \tag{2.13}$$

$$cfm = \frac{1}{k\Delta t + c} \tag{2.14}$$

where Δt is the simulation step, k and c are the spring constant and the damping constant respectively [69].

We set up a mass-spring-damper system to evaluate equation (2.13): the motion of the object is recorded with different ERP and CFM values, then k and c values of the system are determined by system identification. Figure 2.12 shows the resulting k and c according to different CFM values; the experimental results match the theory.

Determinism A physics engine is deterministic when it behaves the same each time it is "replayed" from its original state. Determinism is important in simulation, especially in scientific research where reliability is one of the foundations. However it is known that a major limitation of current realtime physics engines is the lack of determinism. First, the floating calculation may result differently in different platforms. Second, random values are required to avoid singularity in the simulation. Last but not least, random sampling is introduced into simulation for fast approximation, for example, random contact points approximate the contact surface.

We tested the determinism of ODE with a small experiment: let two same boxes fall and collide to the ground at the same time with same initial state (see Figure 2.13).

If simulation is deterministic then the behavior of these two boxes will be the same. The results show that they behave the same at the beginning, but change behavior after a while (see Figure 2.14).

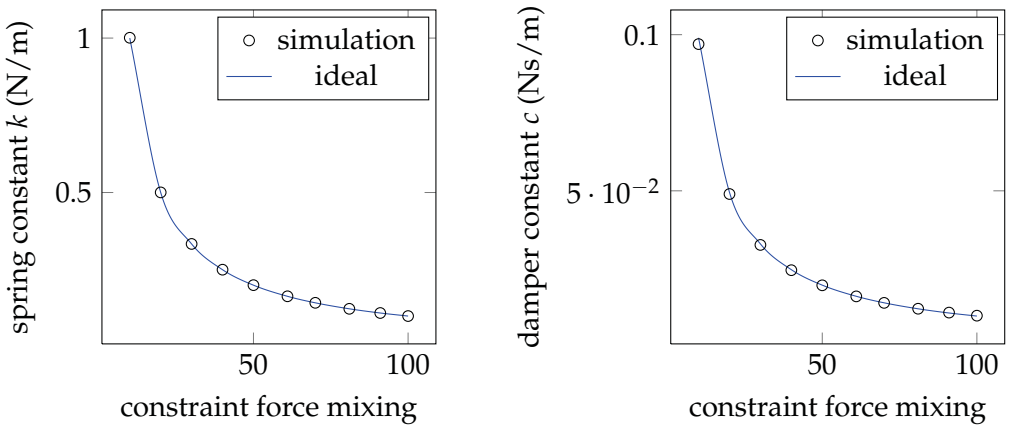


Figure 2.12: The simulated mass-spring-damper system according to CFM (and ERP=0.01).

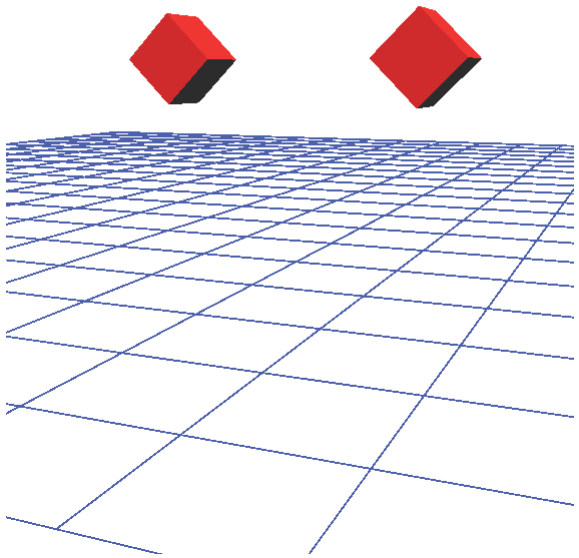


Figure 2.13: Setup of determinism test: two same boxes fall and collide to the ground at the same time with same initial state.

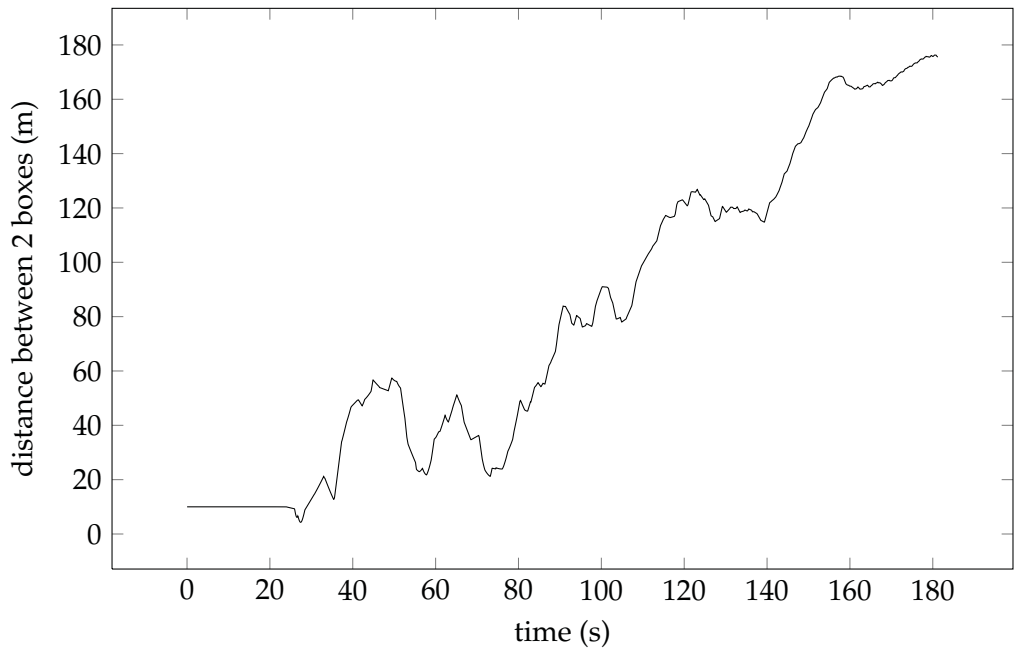


Figure 2.14: Determinism test: the distance between two boxes changes due to the lack of determinism in ODE.

Efficiency The efficiency of a physics engine is also important for robotic simulation, especially when using evolution and machine learning. Because collision detection and solving constraints are the most time consuming parts in a physics engine, we used box stacking to test the efficiency of ODE. In this test a set of $1 \times 1 \times 1\text{m}^3$, 1 kg cubes are placed in a stack on top of one another (see Figure 2.15). Note that we disabled the automatic body disabling in this experiment.

The simulation speed for the corresponding number of stacked boxes is illustrated in Figure 2.16. The simulation gets slower and unstabler as more boxes are added. We were not able to handle more than 20 boxes in real time in this computer.

We evaluated ODE with seven experiments. The results show that ODE emphasizes speed and stability over physical accuracy. Its integrator is stable, but not particularly accurate unless the step size is small enough; collision detection is posteriori based method; some models just use approximations, for example, friction cone is approximated by friction pyramid. These artifacts limit the set of physical phenomenon that can be reproduced at a level adequate for robot applications. Last but not least, ODE is not fully determinate, which can cause problems in repeated experiments.

Nevertheless, ODE is the most popular rigid-body dynamic implementation for robot simulation. While using it to simulate robot motion like biped locomotion, we have to carefully choose some parameters. For example, step size, CFM, ERP etc. We also have to keep shortcomings such as friction approximation, in mind, and build our own models.

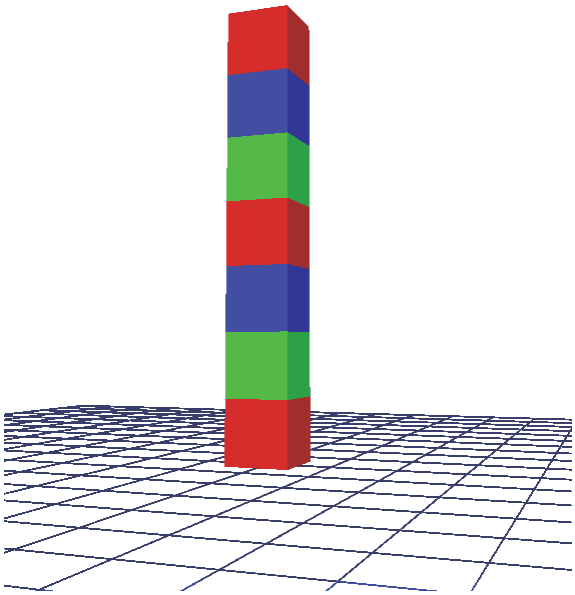


Figure 2.15: Setup of efficiency test: boxes are placed in a stack on top of one another.

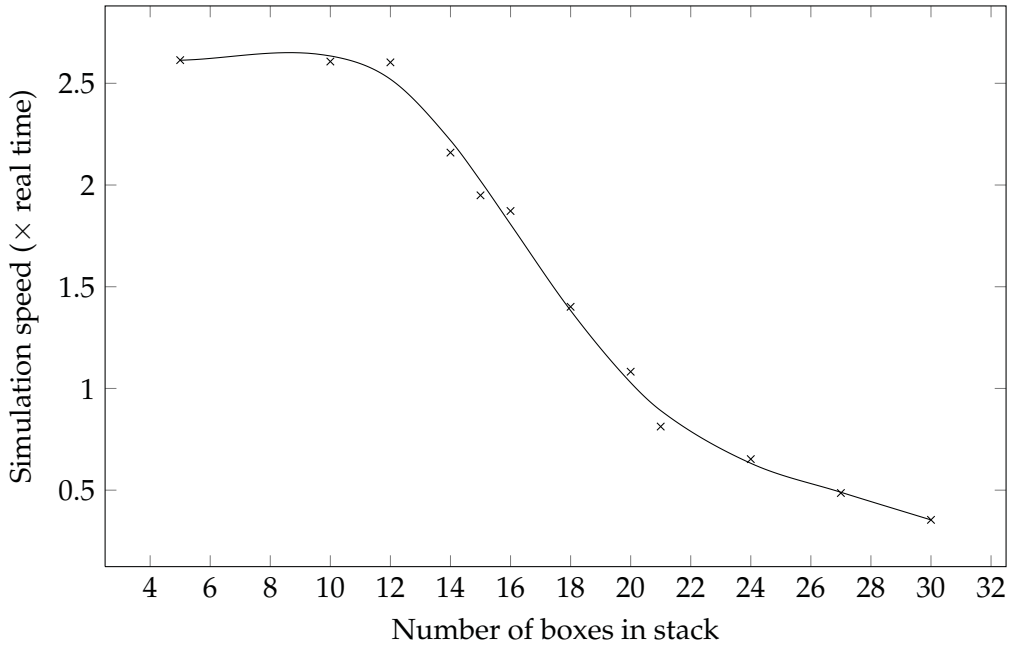


Figure 2.16: Performance of ODE in the box stacking test. The test has been done on a laptop with Core 2 2.0GHz CPU and 4G RAM; the simulation step time is 10 ms.

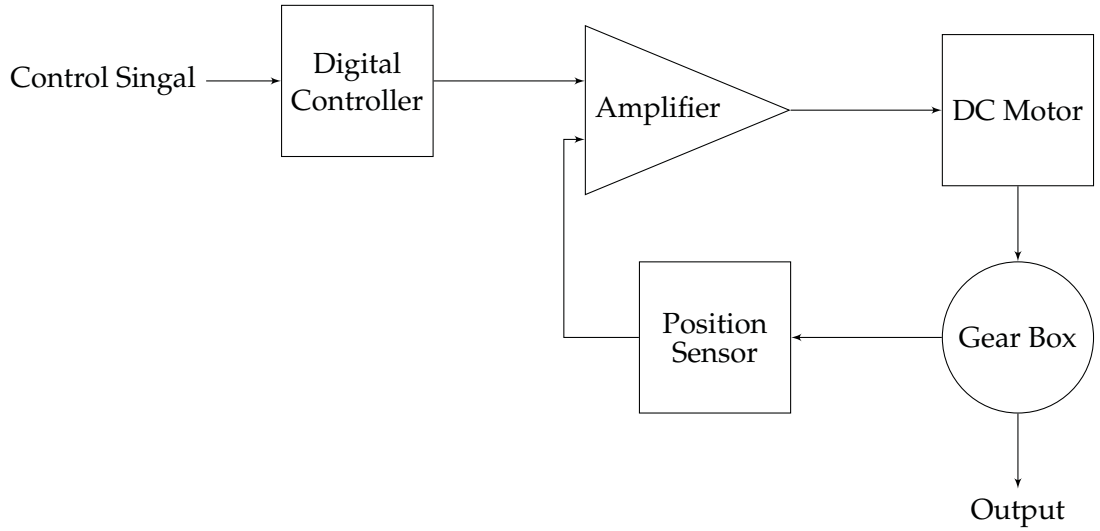


Figure 2.17: A typical DC servo motor system.

2.2.2 Robot Hardware Model

In addition to the physical dynamics model, the utilization of robotic simulation strongly relies on the level of correspondence between the characteristics of real and simulated hardware. In other words, the working mechanism of hardware in real robots should be modeled and simulated properly. Table 1.2 lists the specification of the real NAO robot and simulation model in the SimSpark. We discuss them in detail and evaluate the simulation models in this section.

Joint Motor

The motion of the humanoid robot usually is realized with joints, which are driven by servo motors. Therefore the joint-servo model is important in simulating a humanoid robot.

Servo motors have been around a long time and are used in a wide variety of applications. The basic design of the servo motor is the same no matter the size or manufacturer. Servos are often controlled by setting a desired position, and can only generate a maximum amount of power to achieve that position. As shown in Figure 2.17, a servo motor mainly consists of a DC motor, gear system, a position sensor which is mostly a potentiometer, and control electronics. The DC motor is connected with a gear mechanism which provides feedback to the position sensor. From the gear box, the output of the motor is delivered via servo spline. The typical specifications of servo motors are torque, speed, weight, dimensions, motor type and bearing type.

In simulation, the robot joints are implemented with hinge constraint, which restrict two angular degrees of freedom in a fixed position, so the body can only rotate around one axis. The range limits of joints are implemented with additional constraints.

Because the parameters of the servo controller are unknown, it is difficult to simulate exactly how the servo motor works in area such as, applying a force to a body to achieve a desired position. ODE provides a simple model of real life servos. It has two parameters: a desired speed and the maximum force that is available to reach that speed. The motor brings the body up to speed in one step; and provides force that is not more than is allowed.

This method sees one step into the future to work out the correct force. There are no extra parameters because motors are actually implemented as constraints. This makes joint motors more computationally expensive than computing the forces yourself, but they are much more robust and stable, and far less time consuming to design with.

But this is an idealized motor. There is no motor that works like this in reality. Furthermore, some aspects like power consumption and temperature regulation, are missing but are also important for robotics.

NAO has two types of motors and different types of speed reduction ratios for each of the motors. Different combinations of motors and reduction ratios are used for different joints (see appendix 1.1 for details). The motors used for the NAO joints are Maxon coreless brush DC motors. These motors are controlled by setting desired positions and stiffnesses. The stiffness denotes how strong the joint is. The motor is off when the stiffness is 0; it is most powerful when the maximum stiffness (1.0) is set.

Because the input of the motor in the simulation is the target speed only, a proportional derivative (PD) controller can be implemented for simulated NAO joints:

$$\dot{\theta}_d(t) = K_p e(t) + K_d \dot{e}(t) \tag{2.15}$$

$$e(t) = \theta_d(t) - \theta(t) \tag{2.16}$$

where $\dot{\theta}_d(t)$ is the desired speed, i.e. the output of the PD controller; K_p and K_d are the proportional and derivative parameters of the PD controller; $\theta_d(t)$ and $\theta(t)$ are the desired position and the current position. Note that $K_p = 0.7/t_\Delta$, $K_d = 0.7$ are used in the following tests. Because the parameters of the PD controller affects the results, we chose them by optimizing the performance of the PD controller.

There are 4 aspects of joint motor which have to be evaluated in the experiments, including:

- speed limitation, the maximum speed;
- delay between the sending command and receiving sensor data;
- maximum torque;
- joint limitation (mechanical stop).

We evaluated the performance of simulated joints by executing a predefined trajectory. Two common input signals in system identification are used here: step function and sinusoid function.

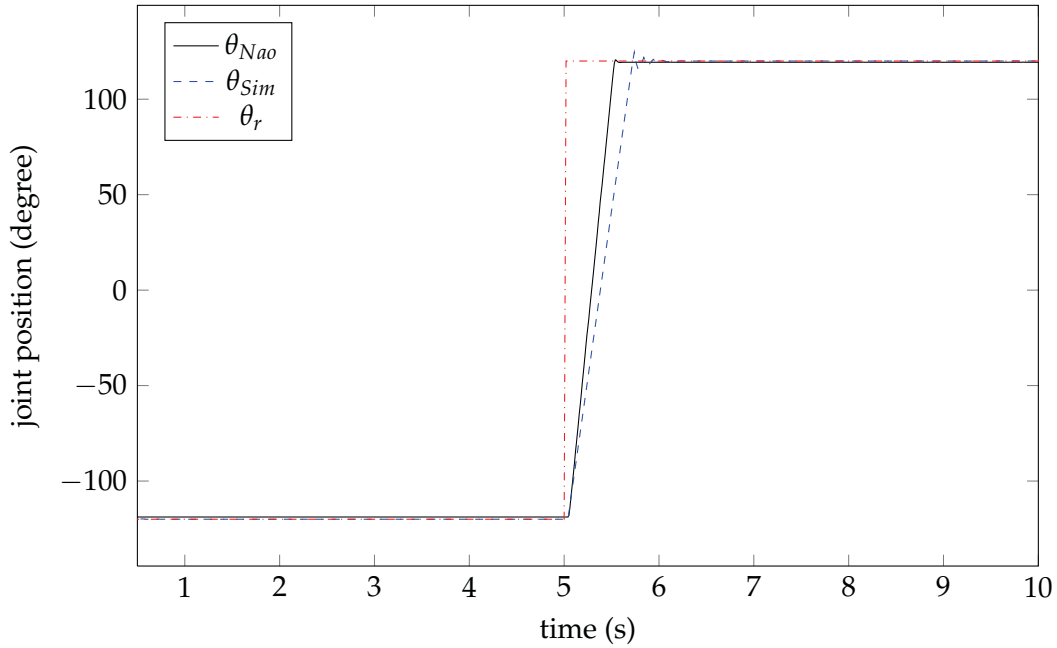


Figure 2.18: The response of the HeadYaw joint to step input in the real robot and simulation. θ_r is the input angles, θ_{Nao} and θ_{Sim} are output from real robot and simulation respectively.

The step response is related to rise time, overshoots, static gain, etc. We chose the range of the joint as step amplitude, i.e. the step input is:

$$\theta(t) = \begin{cases} \theta_{min}, & t < t_0 \\ \theta_{max}, & t \geq t_0 \end{cases} \tag{2.17}$$

where t_0 is the time of step; and $[\theta_{min}, \theta_{max}]$ is the joint range.

Figure 2.18 is the experiment result. The real servo responses was faster than the simulated one. It may be affected by speed limitation or maximum torque. There are small shocks in the simulation only, which may be due to the mechanical stop. The difference between result in real servo and the simulation gives us hints to improve the servo model in the next section.

The input signal $\theta(t)$ of sinusoid is given by

$$\theta(t) = k\theta_{min} + (1-k)\theta_{max} \tag{2.18}$$

$$k = (\sin(wt) + 1)/2 \tag{2.19}$$

where w is the angular frequency which specifies how often oscillations occur; the amplitude gain is the range of joints. Figure 2.19 shows the experiment results of different w values. There are two noticeable differences: 1) there is a delay at the beginning of the motion in the real robot, but almost no delay in the simulation; 2) the performance of the

real servo is better than the simulated servo, i.e. follows requested trajectory closer.

Figure 2.20 compares the maximum speed of the real servo and the simulated servo in the experiment. The maximum speed of the real servo is faster than the simulated one, this is Type I error. This is because the fastest speed of the real servo is faster than no load speed and nominal speed in the specification, and the nominal speed was set as the maximum speed in the simulation.

The evaluation was done with one motor, i.e. HeadYaw. It is sufficient for NAO, because all the motors NAO are DC motors (see table A.1),

Sensors

Although the NAO robot in the SimSpark is equipped with the corresponding simulated sensors of the real robot, they are different from the real ones.

In the real NAO, two identical video cameras are located in the forehead. They provide images with 640×480 pixels at 30 frames per second. They can be used to identify objects in the visual field such as goals and balls. The image processing was not needed in the simulation up to now, since the robot receives abstract vision percepts from the simulator directly. The vision percepts of each object contains distance and angle relative to the camera coordinates. The real robot has to calculate related percepts by image processing.

Using percepts, the real robots, as well as the simulated robots, have to calculate higher levels of information, for example, for localization and speed of objects. The vision percepts in simulations provides more abstract information compared to real images; it contains information about the head, body, hands, and feet of other robots. Therefore, it is possible that the robot can better recognize the situation in the environment and perform more complex tasks.

NAO is equipped with sonar which allows it to estimate the distance to obstacles in its environment. Because of the powerful virtual vision sensor, there is no sonar in the simulated NAO due to lack of necessity.

There are four force sensitive resistors (FSR) installed in each foot of the NAO, these sensors measure the pressure applied. The simulated NAO has one force sensor installed in each foot, which can measure not only the magnitude but also direction of the pressure applied.

The inertial unit is located in the chest of the NAO with its own processor. The output data enables an estimation of the chest speed and attitude (yaw, roll). The inertial unit contains a two axis gyrometer and a three axis accelerometer. The NAO in the SimSpark has a three axis gyrometer and a three axis accelerometer; the sensor data are calculated from the physic engine directly.

The noise in the sensor data raises the uncertainty of the system, and causes problems in transferring the program from simulation to reality. However, the noise of the sensors is not modeled except for the vision sensors.

Besides of the differences mentioned above, some devices like LEDs, loudspeakers are missing in simulation, because they are not necessary for a soccer game.

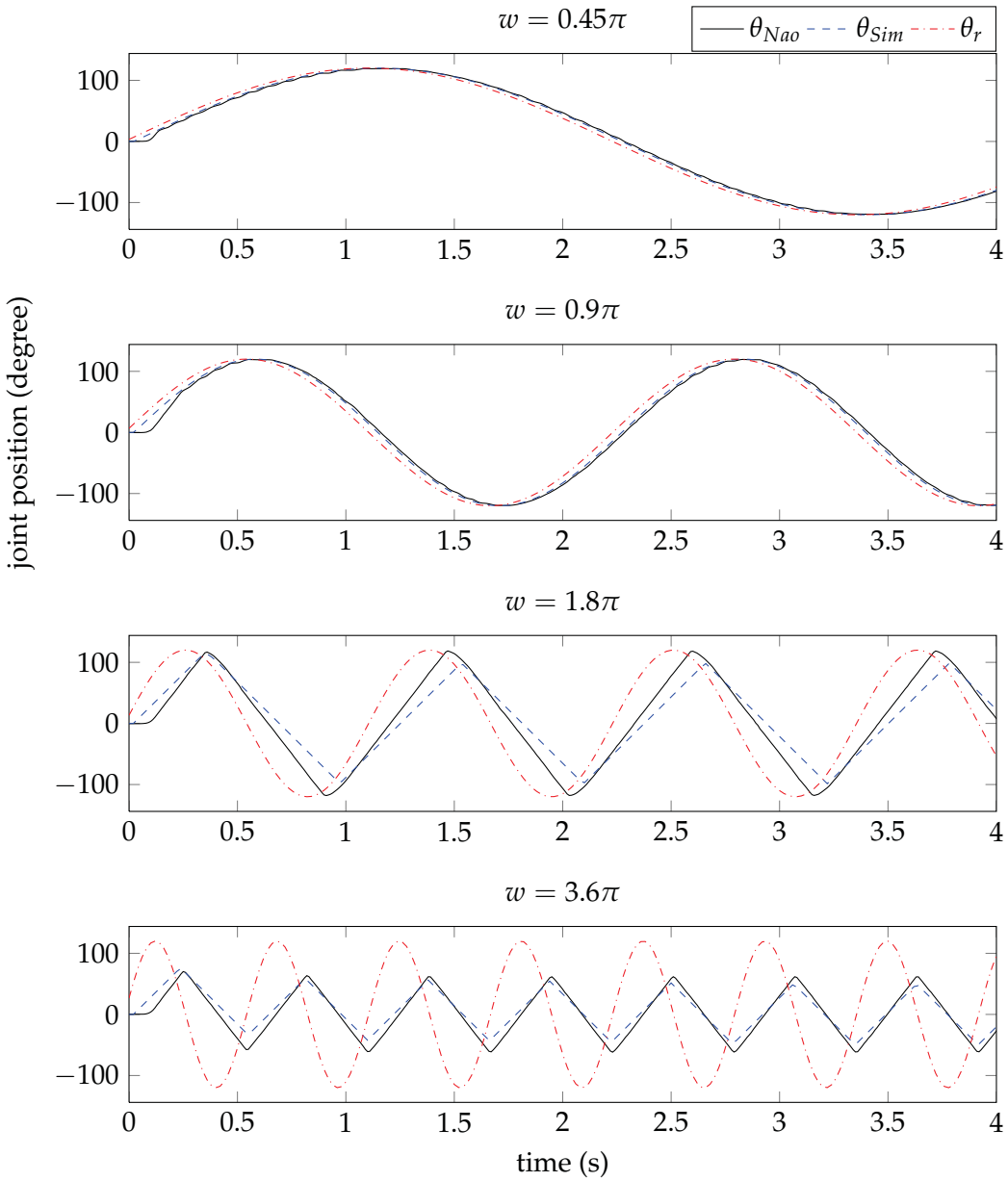


Figure 2.19: The sinusoid response of the HeadYaw joint in the real robot and the simulation with different w values in the equation (2.18). θ_r is the input angle, θ_{Nao} and θ_{Sim} are output from real robot and simulation respectively.

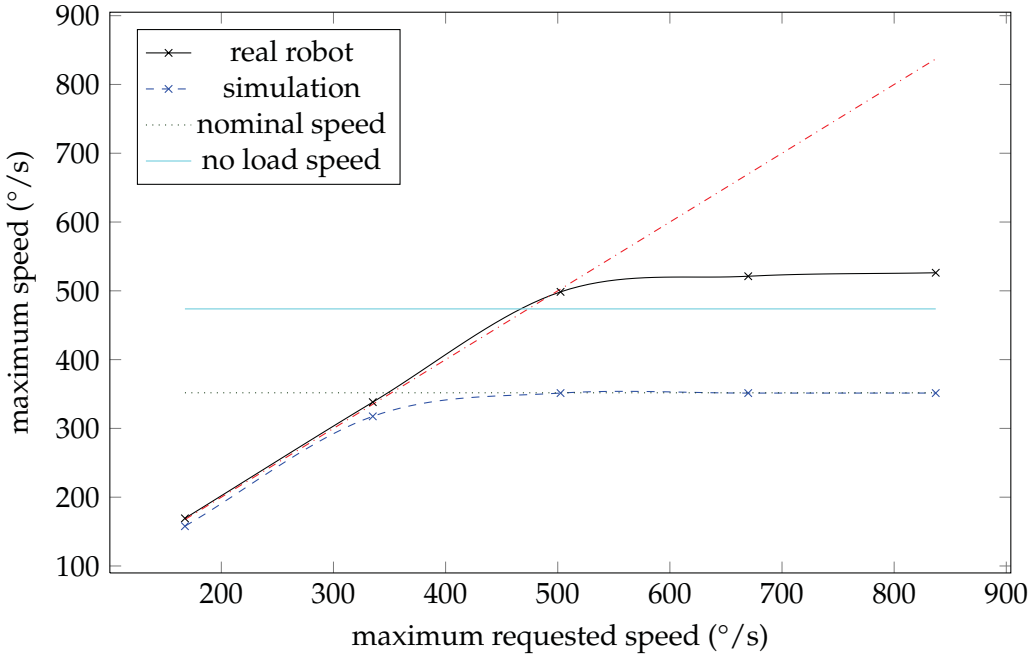


Figure 2.20: The maximum speed of the HeadYaw joint.

2.3 Results Evaluation

Collecting data from reality is so important in the simulation evaluation, that we use both robot sensors and external sensors to collect data.

Some data can be measured directly by the robot's sensors, such as joint data. But some important data can not be measured by the robot's sensors, such as the poses of robot bodies. The kinematic chain of the robot can be calculated by joints sensor data and inertial sensor data. This data can be used to evaluate the simulation. However, some sensors on the NAO robot are unreliable, such as the inertial sensor.

2.3.1 RGB-D Sensor Based Motion Capture System

The Motion capture system is useful for collecting a robot's motion data, however it is traditionally expensive and difficult to use. In this paper, we use a low cost RGB-D sensor (Kinect) to collect motion data from the robot. The RGB-D sensor can not only provide images but also the depth of each pixel. Kinect offers 640×480 pixels RGB-D data at 30 Hz (see Figure 2.21). In order to get cloud points in 3D space, the raw depth data has to be extracted. The raw sensor values returned by the Kinect's depth sensor are not directly proportional to the depth. Instead, they scale with the inverse of the depth. The relationship that maps the depth data (x, y, Z), where Z is the depth value of the pixel (x, y), to the points in the physical world with coordinates (X, Y, Z) is called inverse

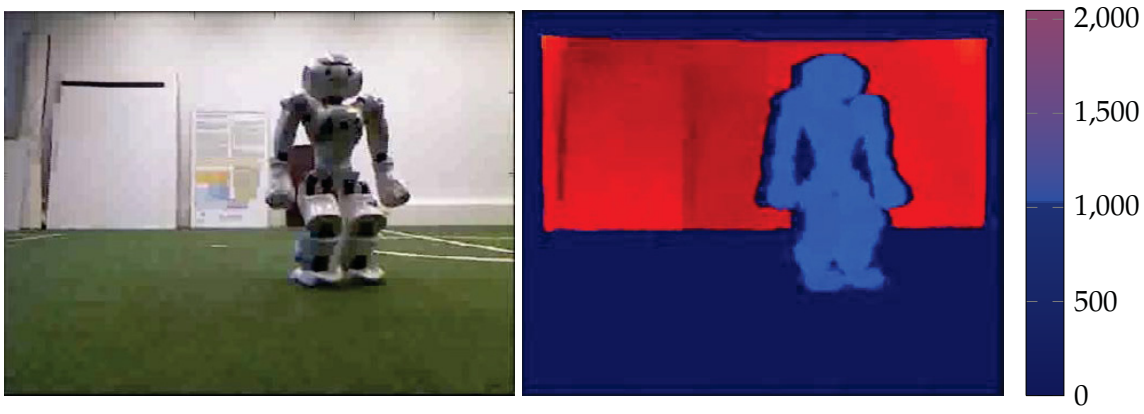


Figure 2.21: Raw data from the RGB-D sensor: left image shows the RGB image data, and right image shows the depth data. Each pixel in depth data is an integer in range (0, 2047).

projective transform:

$$X = \frac{(x - c_x)Z}{f_x} \tag{2.20}$$

$$Y = \frac{(y - c_y)Z}{f_y} \tag{2.21}$$

where c_x, c_y, f_x and f_y are parameters of the camera's intrinsic matrix. After transforming data to 3D space, we get the points data that represents the robot (see Figure 2.22).

In order to use the RGB-D data for evaluation, we implemented the simulated RGB-D sensor, i.e. the points of the robot in simulation are generated by OpenGL. Firstly, we read the depth data of each pixel from the depth buffer, and then transformed the data to points in 3D space with the equation (2.20). Figure 2.23 shows the final result of the simulated RGB-D sensor.

Two technical problems needed to be solved for comparing two point clouds in the time series: 1) the time has to be synchronized; and 2) the poses of the cameras in simulation and reality relative to the robots have to be matched.

For time synchronization, a connection between the robot and the motion capture system is established. The robot can provide extra information such as starting point of motion. In this way, the captured data can be synchronized with this information.

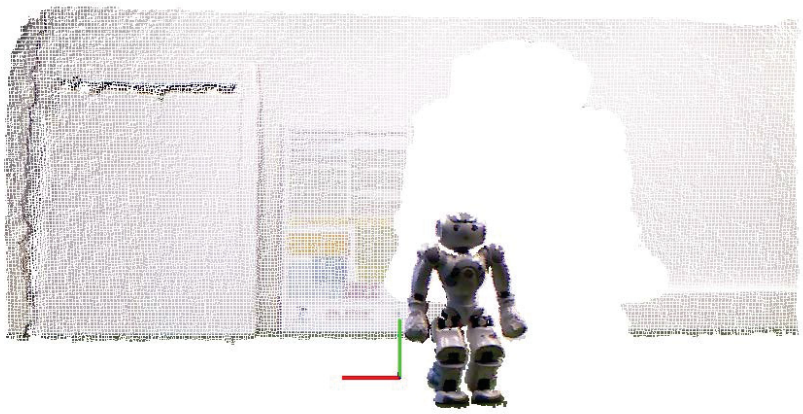


Figure 2.22: The RGB-D data from the Kinect is transformed to points cloud data in 3D space.

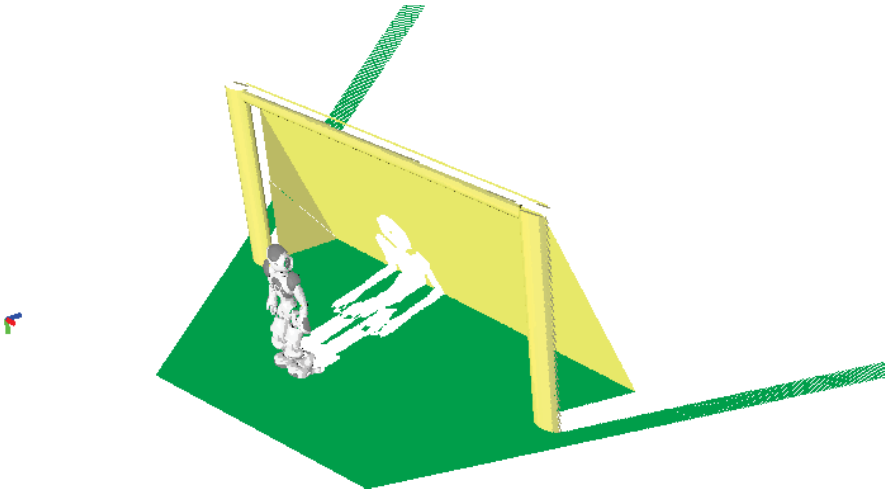


Figure 2.23: Points cloud data generated by the simulated RGB-D sensor.

We have

$$t_0 + \Delta t_1 = t_1 \tag{2.22}$$
$$t_2 - \Delta t_2 = t_1 \tag{2.23}$$
$$t_1' - t_1 = t_d \tag{2.24}$$

where t_0, t_1, t_2 are time in the motion capture system clock, t_0 is the time when sending message, t_1 is the time when message arriving, and t_2 is the time when receiving message; t_1' is the time in robot's clock; $\Delta t_1, \Delta t_2$ are network delay, we assume they are equal, then

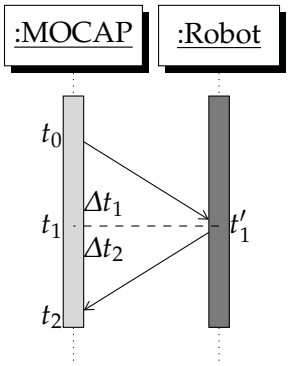


Figure 2.24: Sync of the clocks between the robot and the motion capture system.

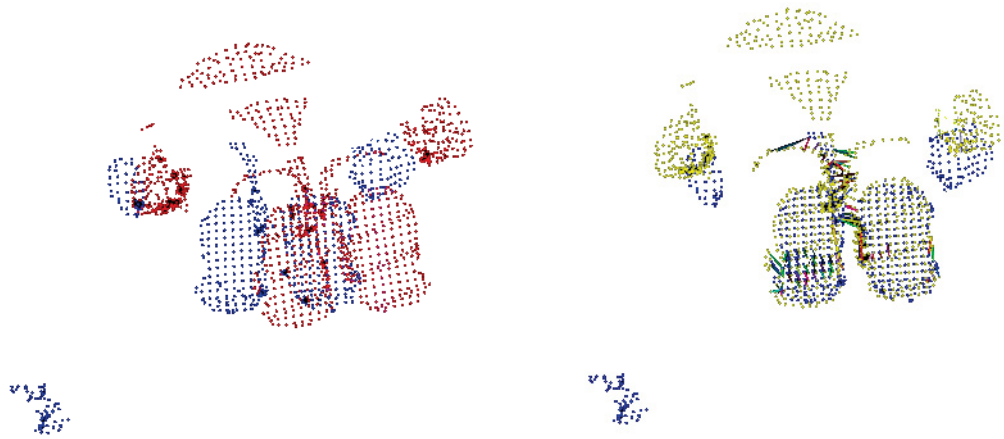
the difference of clocks between motion capture system and robot t_d can be calculated:

$$t_d = t_1 - t_1' = \frac{t_0}{2} - t_1' + \frac{t_2}{2} \tag{2.25}$$

For calibrating poses, we can place the camera carefully and measure the pose between camera and robot. Each time we have to place the robot carefully, this is inconvenient for repeating experiments. We solve this problem by finding a camera pose in the simulation which matches the camera pose in the reality. This is done by point cloud registration, which finds the relative positions and orientations of the separately acquired views of cameras in a global coordinate framework, such that the intersecting areas between them which overlaps perfectly. The Iterative Closest Point (ICP) algorithm is commonly used to minimize the difference between two clouds of points. We use the Point Cloud Library (PCL) [67] for this task. Figure 2.25 shows the process of points registration, and Figure 2.27 shows the results of points registration.

To use the data, we have to filter the background data out, since the captured data contains not only the robot but also the background. We can filter the background data out with a Pass Through Filter [67], which cuts off the values that are outside a given range (see first step in Figure 2.26). But Pass Through Filter is not able to remove the data from the ground. We can put the Kinect on the ground and set the direction parallel with the ground. Because the laser projected by the Kinect will not reflect the ground in this situation, the ground will not be captured (see Figure 2.22). Additionally, outliers, e.g. noisy measurements, can be removed from a point cloud dataset using statistical analysis techniques. For each point, we compute the mean distance from it to all its neighbors. By assuming that the resulted distribution is Gaussian with a mean and a standard deviation, all points whose mean distances are outside an interval defined by the global distances mean and standard deviation can be considered as outliers and trimmed from the dataset.

For further processing, we also reduced the number of points by using a voxelized grid approach (see second step in Figure 2.26). A 3D voxel (i.e., 3D box) grid is created over the input point cloud data. Then, in each voxel, all the points present will be ap-



(a) Initial position of source (red) and target (blue) point clouds. Blue points in the bottom left corner are outliers.

(b) Registration using Interactive Closest Point: target points are in blue, intermediate results are in yellow, and lines indicate correspondences.

Figure 2.25: Points Registration between data from reality (as target) and simulation (as source).

proximated (i.e. downsampled) with their centroid. This approach is a bit slower than approximating them with the center of the voxel, but it represents the underlying surface more accurately.

And finally, we compared two sequences of point clouds for evaluation, see Figure 2.27.

In mathematics, the Hausdorff distance measures how far two subsets of a metric space are from each other. Let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be two cloud of points in 3D, we define the Hausdorff distance $\mathrm{d}_H(\boldsymbol{X}, \boldsymbol{Y})$ by

$$\mathrm{d}_H(\boldsymbol{X}, \boldsymbol{Y}) = \max\{\sup_{x \in \boldsymbol{X}} \inf_{y \in \boldsymbol{Y}} \mathrm{d}(x, y), \sup_{y \in \boldsymbol{Y}} \inf_{x \in \boldsymbol{X}} \mathrm{d}(x, y)\} \tag{2.26}$$

$$\text{where, } \mathrm{d}(x, y) = |x - y|, \tag{2.27}$$

sup represents the supremum and inf is the infimum.

Informally, two cloud points are close in the Hausdorff distance if every point of either set is close to some point of the other set. The Hausdorff distance is the farthest point of a set that can be to the closest point of a different set. In computer graphics the Hausdorff distance is used to measure the difference between two different representations of the same 3D object, particularly when generating level of detail for efficient display of complex 3D models [12].

However, the Hausdorff distance may be dominated by one point (see Figure 2.28). It

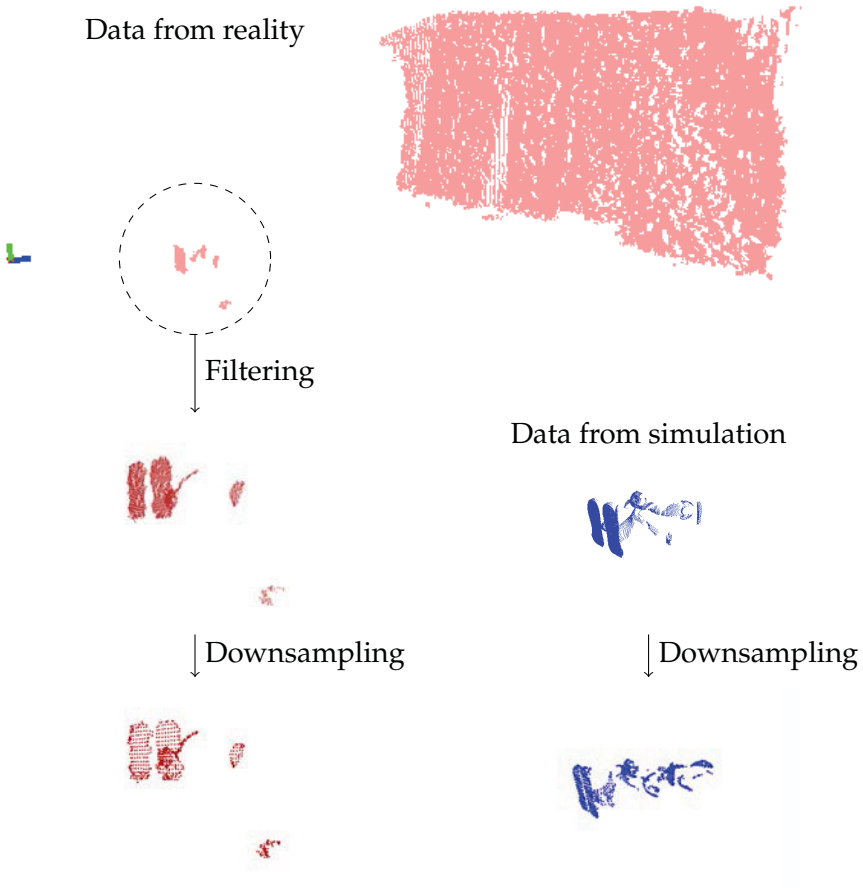


Figure 2.26: Point cloud perprocessing: firstly the background data are filtered, and then the data is downsampled.

means that it is not good to measure the difference in the transformation of two points set when the data is noisy, so we use the root mean square error (RMSE) to the nearest neighbor point of each point instead of the maximum one, e.g.:

$$d_S(\boldsymbol{X}, \boldsymbol{Y}) = \max\{\frac{1}{n_{\boldsymbol{X}}} \sum_{x \in \boldsymbol{X}} \inf_{y \in \boldsymbol{Y}} d(x, y)^2, \frac{1}{n_{\boldsymbol{Y}}} \sum_{y \in \boldsymbol{Y}} \inf_{x \in \boldsymbol{X}} d(x, y)^2\} \tag{2.28}$$

Where $n_{\boldsymbol{X}}$ and $n_{\boldsymbol{Y}}$ are the number of points in $\boldsymbol{X}$ and $\boldsymbol{Y}$. In our case, $\boldsymbol{X}$ and $\boldsymbol{Y}$ are points data from simulation and reality respectively, and the value of $d_s(\boldsymbol{X}, \boldsymbol{Y})$ indicates the difference between simulation and reality.

Figure 2.29 illustrates the whole evaluation process of the RGB-D sensor motion capture system. Using the Kinect as the motion capture system has advantages: it is not only cheap and easy to set up, but also supports full body motion capture. On the other hand,

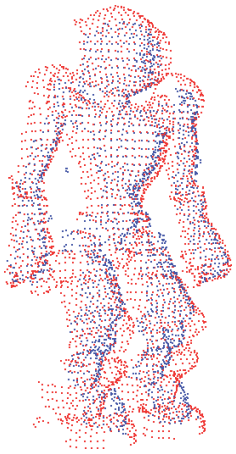


Figure 2.27: Comparing points data from Kinect (red) and simulation (blue).

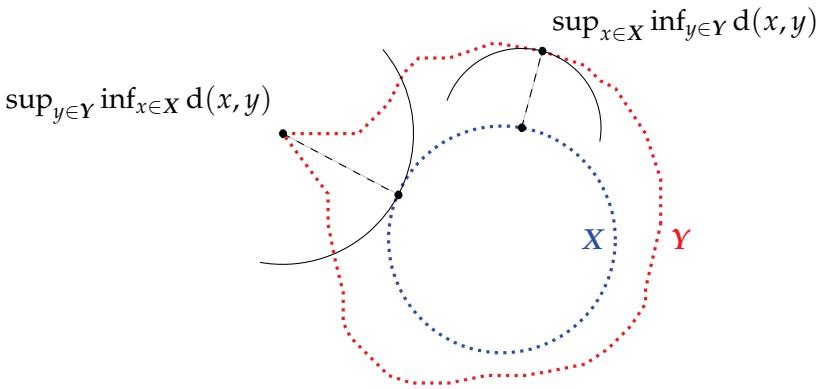


Figure 2.28: Calculation of the Hausdorff distance between the green points set X and the blue points set Y. The result is dominated by one peak point in set Y.

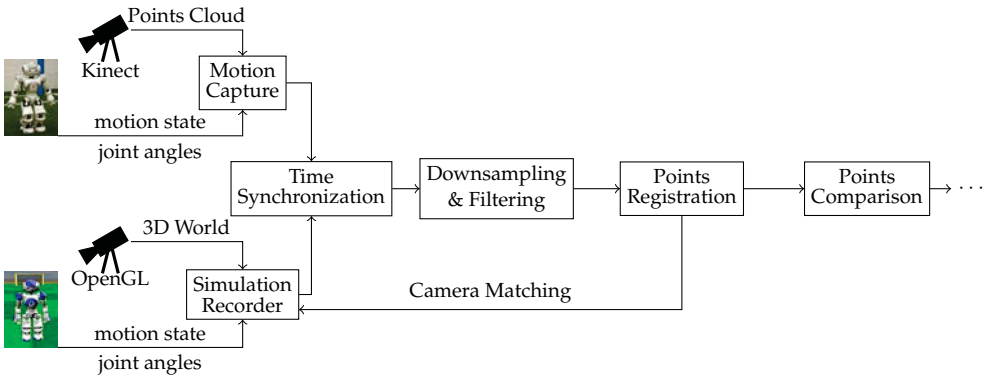


Figure 2.29: Kinect motion capture system for evaluating simulation

due to the cheap price of this system, it has disadvantages when it comes to precision and frame rate. The Kinect works in 30 frames per second, so it can not be used in high speed situations. The noise of data from the Kinect depends on the distance between the object and the Kinect. It is $\pm$1.5 mm at 50 cm and $\pm$50 mm at 5 m.

2.3.2 Motion Evaluation

Static Motion Static motion is developed for demonstrations or special purposes, because it is easy to create and test. Keyframe is one technology used to create static motions, it defines all the joint positions of the robot, and the motion of the robot is interpolated by a sequence of keyframes, for more details see section 4.1.

In the NaoTH, several motions are Keyframe based, such as standing up, goalie jumping, etc. The same Keyframe can not be used with both the real robot and simulation. For example, we created fast standing up in the simulation, which can not be applied to the real robot. Standing up of the real robot also fails in the SimSpark.

Dynamic Motion For some complex tasks, such as biped walking, static motion is not enough. Advanced algorithms are used to take advantage of the robot dynamics. Such dynamic motions are more complex. The simulation results of dynamic motion can be used to evaluate the simulation.

Walking is the most important skill in the RoboCup teams today. Both simulation and standard platform teams have invested a lot effort into walking. The walking speed in the Simulation League is much faster than in the Standard Platform League.

We analyzed the log files from the Simulation League, and got the speed of the top teams from the RoboCup 2011[1]. During the analysis, we extracted all the positions of robot's body in one game and calculate the speed in the horizontal plane. Because the robot may fall down or there may be some unintended motion in short time frame, we dropped all the data with low height and passed it through a low pass filter.

Figure 2.30 compares the teams in the Simulation League with the top teams in the Standard Platform League. Although teams in the Standard Platform League have improved a lot, when compared to the walk provided by the manufacturer Aldebaran, they are still much slower than the simulation teams. Note that the speeds of the Standard Platform League come from the publications of the top teams [17, 32, 77]. These are the maximum forward walking speeds which can be achieved in the experimental environment. The fast speed in games of the Standard Platform League in RoboCup 2011 is approximately 0.2 m/s - 0.25 m/s.

In the NaoTH, we have a much faster walk in the simulation than in the real robot, while using the same algorithm with different set of 26 parameters. We used the evolutionary algorithm to optimize our walking parameters in the SimSpark. We can almost achieve the fastest speed in the Simulation 3D League; but our real NAO is much slower. The robot in the simulation can walk slowly with the parameters from the real robot. All

[1]The logfiles of RoboCup 2011 are downloaded from `http://simspark.sourceforge.net/logs/RoboCup2011/`

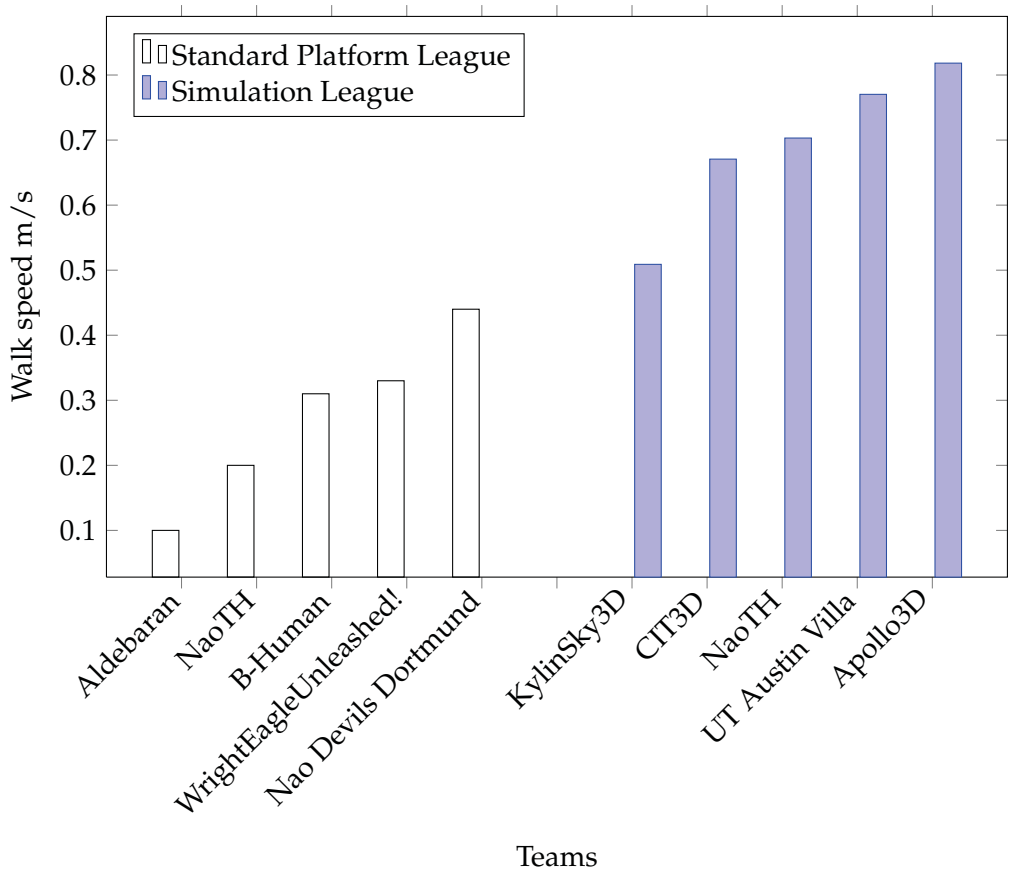


Figure 2.30: Walk speed of teams from the Standard Platform League and the Simulation League. The data of the Standard Platform League comes from their publications. The data of the Simulation League are analyzed from logfiles of the RoboCup 2011.

these factors indicate that simulation results are much better than the real NAO, and it is Type II error. This intuitively points out the difference between reality and simulation.

2.4 Summary

In this chapter, we discussed the methods used to evaluate simulation. Then we evaluated the ODE and the SimSpark in model level and results level. We used data from external sensors (i.e. Kinect) as well as robot sensors.

The experimental results show that the reality gap contains both Type I errors and Type II errors. The motions developed in the SimSpark show that it is unable to perform well in reality. The motions developed in real robots also performed badly in the SimSpark. The reality gap was mainly due to the fact that some models contained approximations or

errors, and the parameters in the ODE affects the results. Furthermore, the servo model was too simple, and robot model was not precise enough. Therefore, the robots from the simulation teams ran much faster than teams in the Standard Platform League (see Figure 2.30). The simulation can not successfully repeat the key frame motion which was developed in the real robot.

In next chapter, we investigate how to improve the realism of the SimSpark based on the evaluation results in this chapter.

3 Improved Realistic Simulator for Humanoid Robot

If I have seen further it is by standing on the shoulders of giants.

— Isaac Newton

Designing and implementing a good robot simulator is a difficult and time consuming task, including software architecture, physical dynamics, visualization, robot model formats, network support, etc. Although the evaluation results in chapter 2 have shown that the motion results of NAO in the SimSpark are different from the real robot, the SimSpark has been developed as an open source project by the RoboCup community since 2004. It has many features for multi robots research, e.g. a flexible application framework, configuration by Ruby script, customized scene description language, multi-threads, logfile recording and replaying, etc.

We investigated how to develop a more realistic simulator to simulate the NAO robot, and used it to develop motions for the real robot in simulation later. Two main topics have been investigated: 1) how to model the humanoid robot properly; and 2) how to choose the parameters of simulation. After evaluation, these three items were used to build a loop to improve the realism of the simulator as shown in Figure 3.1.

We decided to improve the SimSpark instead of creating a new simulator. There are two main reasons: 1) reuse open source code to avoid reinventing the wheel; and 2) contribute the results of the simulator to the community.

This chapter is organized as follows: section 3.1, we modeled and implemented new sensors for the NAO robot, and refined the whole robot model; in section 3.2 parameters of the simulator were optimized according to data from the real robot; and section 3.3 evaluates the improved simulator; in section 3.4 a short summary follows.

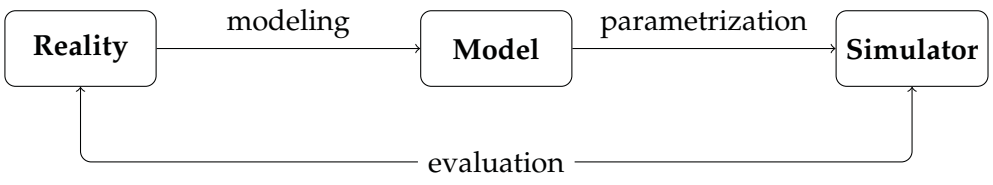


Figure 3.1: Building the simulator: modeling, parametrization, and evaluation.

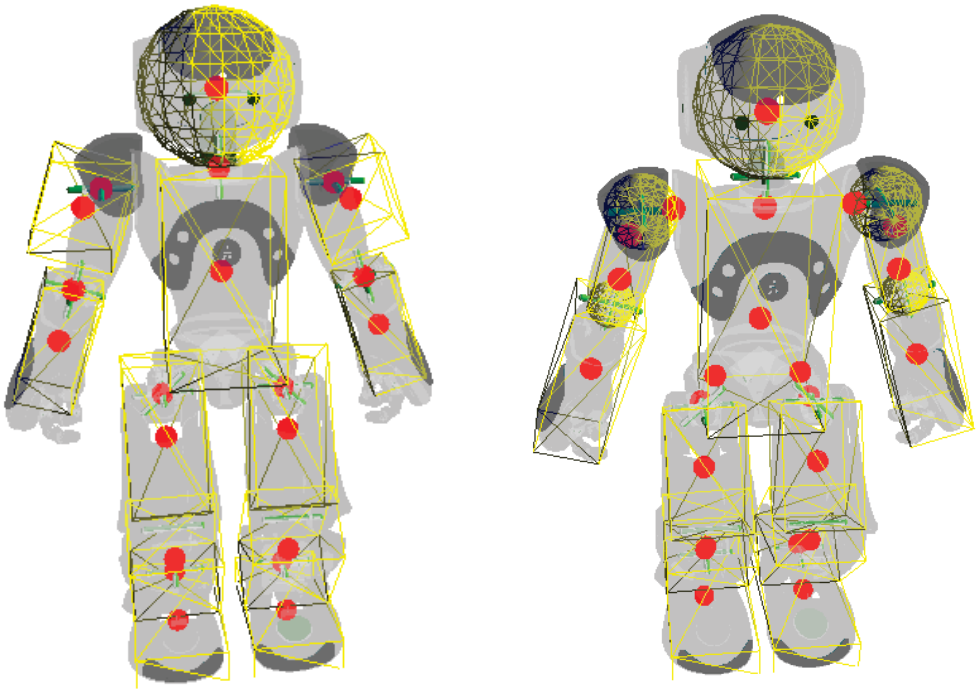


Figure 3.2: The NAO model in the official SimSpark (left) and the refined NAO model (right) according to documentation from the manufacturer. The yellow wire frames are geometries of the robot; the red point indicates the center of mass for each part of the robot; and the green axises denote joints.

3.1 Modeling Humanoid Robot

In the simulation, the robot is modeled as a rigid body system which is an assembly of component parts. Those components include rigid bodies and joints which connect different rigid bodies. A rigid body has eight properties from the point of view of the dynamics. Four of them are constant over time, e.g. mass, center of mass, inertia matrix and shape(*Geometry* in ODE). These properties should be as precise as possible.

The NAO model in the official SimSpark was implemented in 2008 when only the demo version of NAO was available and little information about hardware was available. Therefore, the original NAO model in the SimSpark is outdated.

The problem with this model is that it doesn't provide precise enough information for the physic simulation engine: each component part is a box or a sphere with uniform density; the center of mass is in the geometry center of the body; and the shapes of component parts are not well represented (see the left figure in Figure 3.2).

Aldebaran Robotics gives detailed information regarding mass and inertia matrix in the latest NAO documentation [2]. We refined the simulated NAO model in the SimSpark with this information, including kinecmatics, mass, and inertia matrix. And the shapes

are also refined using the real robot for details. Because the center of mass and the point of reference must coincide in the ODE, we used *Geometry Transform* to reposition the shape. Therefore, all the positions of bodies and shapes had to be re-calculated according to the center of mass. After refining, the NAO model is more accurate and closer to the real NAO (see the right figure in Figure 3.2).

3.1.1 Sensor Modeling and Implementation

Autonomous robots like the NAO usually have many different sensors installed. Missed sensors or bad modeled sensors make simulation results invalid. The sensor difference between the NAO in the SimSpark and reality are listed in table 1.2. We came up with implementing new sensors in the SimSpark in this section. Some implementation, e.g. RGB camera and accelerometer, were included in the official SimSpark repository.

Accelerometer Because the ODE does not provide acceleration of objects, we have to calculate acceleration based on velocity and then transform it to the body's local coordinate. It is important to note that the real accelerometer measures the proper acceleration relative to free fall. As a consequence an accelerometer at rest relative to the Earth's surface will indicate approximately $1\boldsymbol{g}$ upwards. This "gravity offset" should be added. The accelerometer is update as follows:

$$\boldsymbol{a} = \boldsymbol{R}^{-1}\left(\frac{\boldsymbol{v}_1 - \boldsymbol{v}_0}{t_1 - t_0} - \boldsymbol{g}\right) \tag{3.1}$$

where $\boldsymbol{v_0}$ and $\boldsymbol{v_1}$ are global velocity in time t_0 and t_1 respectively, $\boldsymbol{R}$ is the rotation matrix of the body in the global coordinate, and g is the gravity.

RGB Camera We simulated the RGB camera with OpenGL, since the visualization of the SimSpark is build on OpenGL. In particular, the rendering pipeline is:

1. set perspective of camera
2. render the scene
3. read data from buffer
4. encode and send to robot client

The perspective matrix can be set according to the camera intrinsics matrix which can be calibrated by OpenCV [27]. The projection of the points in the physical world into the camera is now summarized by the following formula:

$$\boldsymbol{q} = s\boldsymbol{M}\boldsymbol{p}, \text{where } \boldsymbol{q} = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}, s = \frac{1}{z}, \boldsymbol{M} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \boldsymbol{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{3.2}$$

$\boldsymbol{q}$ is the projection point of $\boldsymbol{p}$ in the viewing coordinate system; and $\boldsymbol{M}$ is the intrinsics matrix of the camera [27], where (f_x, f_y) is the focal length of the camera, (c_x, c_y) is the displacement (away from the optic axis) of the center of coordinates on the projection screen.

In the OpenGL, the projection has two steps, first projecting points into the Clipping Coordinate System, and then transforming it into the Device Coordinate System by viewport:

$$\boldsymbol{q} = \boldsymbol{V}\boldsymbol{T}\boldsymbol{p}, \text{where } \boldsymbol{V} = \begin{bmatrix} \frac{w}{2} & 0 & \frac{w}{2} + x_0 \\ 0 & \frac{h}{2} & \frac{h}{2} + y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

w and h are the resolution width and height, $\boldsymbol{V}$ is the viewport transformation, so we get the perspective projection matrix $\boldsymbol{T}$ as follow:

$$\boldsymbol{T} = \begin{bmatrix} 2\frac{f_x}{w} & 0 & \frac{2c_x - w - 2x_0}{w} \\ 0 & 2\frac{f_y}{h} & \frac{2c_y - h - 2y_0}{h} \\ 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

The non-linear lens distortions (i.e. radial distortion and tangential distortion) are implemented to suggest the pinhole model to be more realistic. The implementation is OpenGL Shading Language (GLSL) [66] based, which benefits from hardware acceleration of the graphic card. Again, the distortion map is generated from standard camera calibration.

We encoded the image data in Base64, because the protocol between the SimSpark and the robot client are ASCII based. The Base64 encoding scheme ensures that the data remains textual, therefore the image data can be transferred without modifying the protocol of the SimSpark. Furthermore, the data can be rendered off screen by non-displayable framebuffer objects in the OpenGL, which provides an efficient performance for multi-robots and high resolution. For realistic images from the environment, we created a simulation scene in our lab, Figure 3.3 shows the results.

Lines in Virtual Vision The virtual vision in the SimSpark makes the robot able to "see" objects as points; but the robot was unable to see lines which are important land markers for self localization in the Standard Platform League. It was also a problem for simulation teams, because the robot rarely saw land markers (goals) after the field size became bigger. We introduce the line percepts for virtual vision. We modeled each line by its two endpoints, and these pair points are provided to the robot if both the endpoints are in the view range of the robot. If only one point or even no point is in the view range, the robot may still see the line. Therefore, we created a virtual image plane; and projected all the lines on this plane. We then calculated the intersections of lines and the boundary of view range and projected the intersections back to the 3D world. These are endpoints of the lines that that robot can see (see Figure 3.4 for illustration).

Lines in virtual vision not only provide more information to the robot in the simulation game, but also are helpful to debug the image processing program together with the simulated RGB camera. We used lines from the virtual vision to check the line percepts

Figure 3.3: The NAO with RGB camera in the SimSpark. The left bottom sub-screen is the robot's view, i.e. the data received by the robot client.

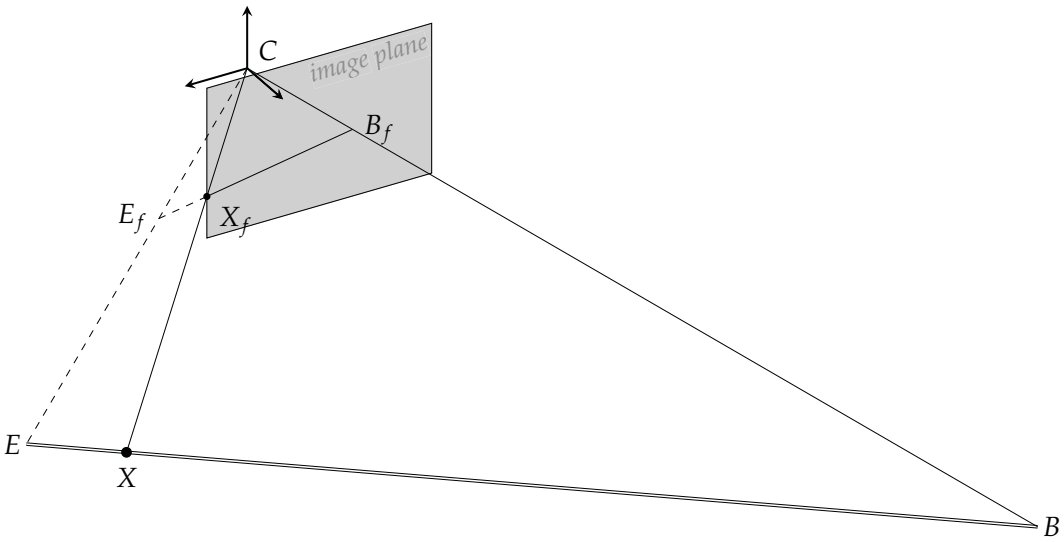


Figure 3.4: Calculating the lines in virtual vision. C is the position of the camera, BE is one line in the 3D world, B_fE_f is the projection of BE on the image plane, X_f is the intersection between B_fE_f and the boundary of image plane, X is the intersection of EB and CX_f, that is the point the robot can see.

from the image processing.

With these sensors, we can use the same modules both in simulation and in the real robot. This enabled us to program the robot in the simulation and transfer the program to real robot without changing code.

3.1.2 Actuator Modeling and Implementation

Besides the sensors, the NAO has twenty-one motor joints as its actuators. The simple motor model is one reason for the unrealistic simulation results.

As mentioned in section 2.2.2, the ODE provides a simple model of real life servos. A PD controller was implemented for the simulated NAO. Based on the evaluation results in section 2.2.2, we modeled additional features in this section, including stiffness control, backlash, and power consumption.

Stiffness The stiffness determines how strong the motor is. The value is from 0.0 to 1.0, 0 means the motor is off and 1 means the motor is running at full power. In the real robot, this percentage is the maximum electric current applied to the motor. Setting the stiffness to 0.5 means that the electric current limitation is reduced to 50%.

For a DC motor, the electric current, I, determines the output torque, τ:

$$\tau = K_\tau I \tag{3.5}$$

where K_τ is the torque constant of the motor. It can be found in the specifications from manufacturer (see table A.2). In the simulation, the maximum torque of the servo can be specified, therefore the stiffness control can be easily implemented by setting the maximum torque of the simulated servo:

$$\tau_{max}(t) = k_s(t) T_{max} \tag{3.6}$$

where $\tau_{max}(t)$ is the maximum torque set in the simulated servo at time t; $k_s(t)$ is the stiffness at time t; and T_{max} denotes the maximum torque of the servo when stiffness is 1.

Backlash Although Aldebaran Robotics uses spur and planetary gears in NAO to have a good back drive ability, there is still noticeable backlash: when the robot is in its initial position (all motors in position 0), the robot is unable to stand in a perfectly vertical position. In mechanical engineering, backlash is the clearance between mating components, sometimes described as the amount of lost motion due to clearance or slackness when reversed and contact is re-established. Without backlash, a system would be subject to overloading and overheating, and then as a consequence, there would be failure of the whole system. The backlash also allows good lubrication of the teeth, which is important. Therefore the backlash is necessary, and is not completely avoidable.

In [31], Aldebaran Robotics said that the angular joint backlash must remain between $-3°$ and $3°$. Backlash decreases the control performance. This problem is a well known

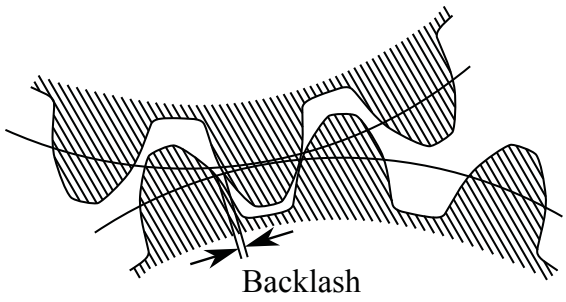


Figure 3.5: Backlash between gears.

in robotics. We simulated the backlash by using the dead band model [18]. The dead band in the backlash is a region of input motion in a mechanism which results in no appreciable output motion. In the simulation, the motor does not apply any torque while the position is in the dead band. The mathematical description of this dead band model is

$$\tau_m(t) = \begin{cases} \tau_{max} & \text{if } \dot{\theta}(t) > 0 \text{ and } \theta(t+1) \geq \theta(t) + \theta_+ \\ & \text{or } \dot{\theta}(t) < 0 \text{ and } \theta(t+1) \leq \theta(t) + \theta_-; \\ 0 & \text{otherwise.} \end{cases} \tag{3.7}$$

where $\tau_m(t)$ denotes the actual maximum servo torque at time t; $\theta(t)$ and $\dot{\theta}(t)$ are the joint angle and velocity at time t; θ_+ denotes the clearance on the "positive" side of the dead band; and θ_- denotes the clearance on the "negative" side of the dead band.

Power Consumption Another important aspect besides the motor's performance is its power consumption: how much energy does it cost to run. The robot is powered by a battery with limited energy, and has to walk during the game: half a game is 10 minutes. An even more important factor in energy consumption is that the motor can overheat if it consumes too much energy and becomes too hot. In the real NAO, Aldebaran Robotics estimates the temperature of each motor by integrating electric current, and shuts down the motor when the temperature is too high. Therefore modeling power consuming is important for implementing energy effective motions.

DC motors are based on the following equations:

$$U = U_e + IR \tag{3.8}$$

$$U_e = K_e\dot{\theta} \tag{3.9}$$

where U is the voltage of input, U_e is the back electromotive force (EMF), I is the electric current, R is resistance, $\dot{\theta}$ is the speed, and K_e is the speed constant of the motor.

The value of R and K_e can be found in the specifications from the manufacturer (see table A.2); and the simulation engine provides the value of τ and $\dot{\theta}$; therefore we can

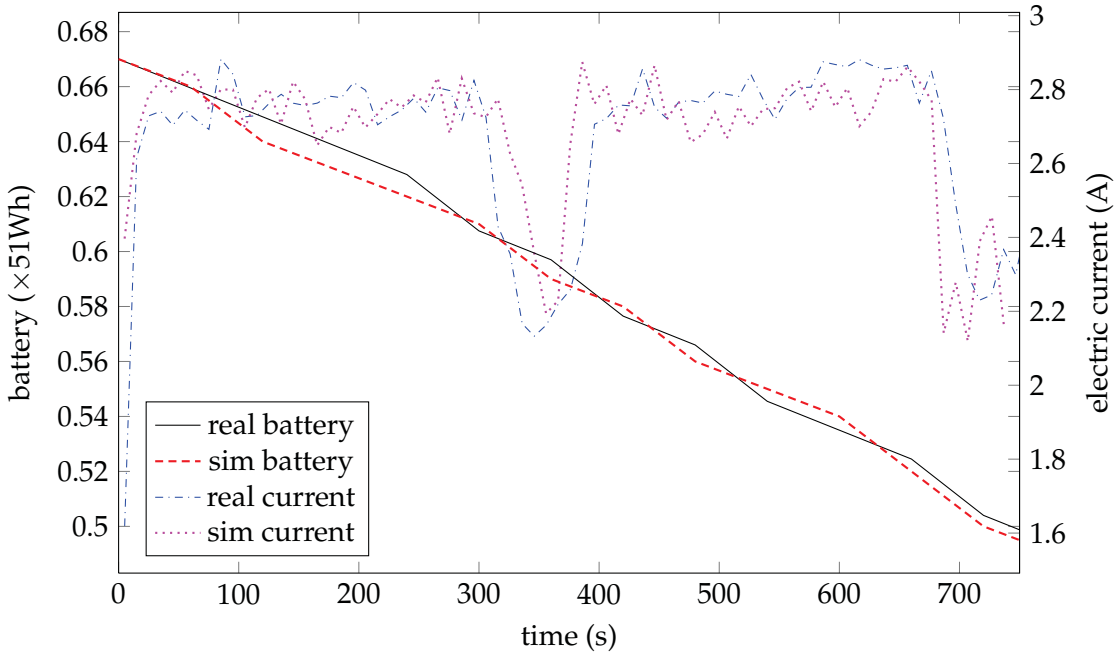


Figure 3.6: Power consumption of the real and the simulated robot in action. The electric current is the summary of all motors.

calculate the power consumption by putting equations (3.5), (3.8) and (3.9) together:

$$P = UI \tag{3.10}$$
$$= U_e I + I^2 R \tag{3.11}$$
$$= \frac{K_e}{K_\tau}\dot{\theta}\tau + \frac{R}{K_\tau^2}\tau^2 \tag{3.12}$$

And the total energy used by the motor is:

$$E = \sum_t P_t \Delta t \tag{3.13}$$

where Δt is the time step of the simulation, and P_t is the power consumed at time t.

Figure 3.6 compares the simulation result of this model to reality. In this example, the robot turns left for 5 minutes, then stands for 1 minutes and then turns right for 5 minutes. This is shown by the change of electric current in the figure. For the overall power consumption, the energy consumed by devices other than motors, e.g. mainboard, CPU, camera, etc. has to be added. It is the power consumption of the robot in an idle state (all motors are off), and measured to be 33 W.

We model the temperature and heat of the motor with the following equations:

$$\Delta Q^+ = I^2 R \Delta t \tag{3.14}$$
$$\Delta Q^- = -\lambda (T - T_e) \Delta t \tag{3.15}$$
$$\Delta Q = \Delta Q^+ + \Delta Q^- \tag{3.16}$$
$$C = \frac{\Delta Q}{\Delta T} \tag{3.17}$$

where T is the temperature of the motor, T_e is the temperature of the environment, but it is the internal temperature of motor, so it is higher than outside and differs from motor to motor, ΔQ^+ is the heat produced by the motor, ΔQ^- is the heat transferred from the motor to the environment, ΔQ is the heat changing, λ is thermal conductivity which indicates the ability of a motor to conduct heat, and C is the heat capacity of the motor, which can be seen as constant. Finally, the temperature of the motor at time $t + \Delta t$ can be solved as:

$$T_{t+\Delta t} = T_t + \Delta T = T_t + \frac{[I^2 R - \lambda (T_t - T_e)] \Delta t}{C} \tag{3.18}$$

In this model, we determined T_e, λ, and C by experiments. It can be formulate as a classic linear regression problem. We rewrite equation (3.18) to:

$$\Delta t x_0 + T_t \Delta t x_1 - \Delta t x_2 = I^2 R \tag{3.19}$$
$$x_0 = C \tag{3.20}$$
$$x_1 = \lambda \tag{3.21}$$
$$x_2 = \lambda T e \tag{3.22}$$

A sequence values of Δt, T_t, and $I^2 R$ can be measured by experiment, therefore the optimum values for x_0, x_1, and x_2 can be obtained by linear least squares method, so the parameters of equation (3.18) are determined. Table 3.1 gives the results of the experiment.

Table 3.1: Parameters T_e, λ and C in the equation (3.18) for different joints of the NAO V3.

Joints	T_e (°C)	λ (W/°C)	C (J/°C)
HipYawPitch	33	0.0158	27.2
HipRoll AnkleRoll	27	0.0158	27.2
HipPitch KneePitch AnklePitch	27	0.0198	43.6

After determining the parameters in the equation (3.18), we can use this model to simulate motor temperature. In Figure 3.7, the simulated temperature is compared with data

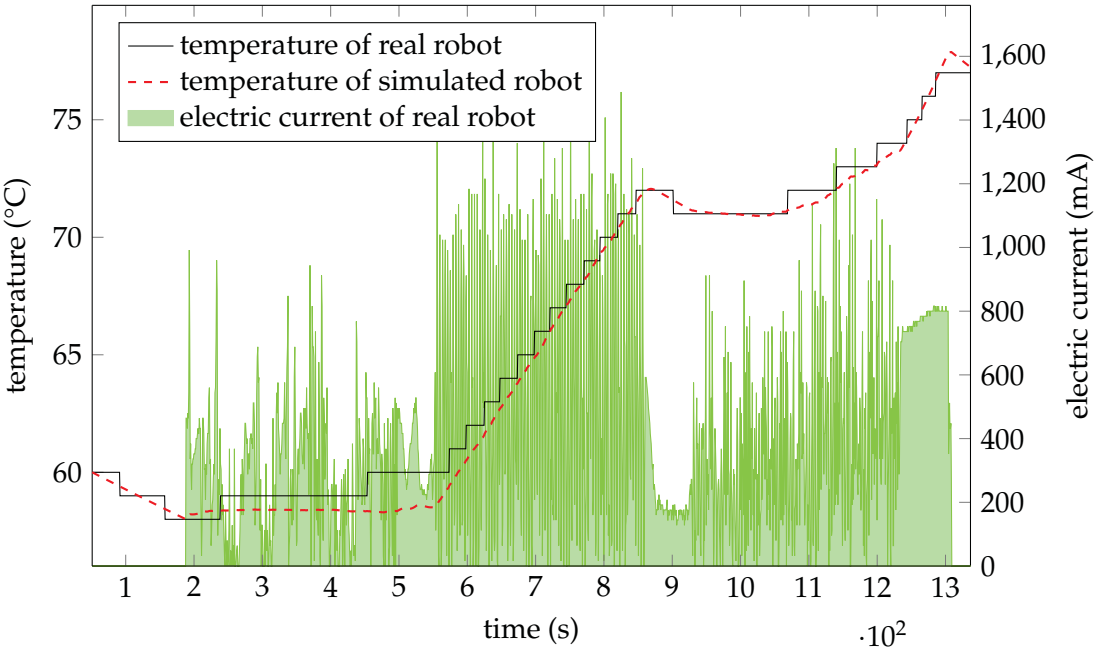


Figure 3.7: The temperature of the (knee pitch) motor in the simulation and the real robot. The green background is the electric current in the real robot. The result shows the simulated temperature is very close to the values of the real robot.

from the real robot.

The whole process of joint simulation is summarized in Figure 3.8: stiffness is simulated by setting the maximum torque of the motor; desired speed of motor is determined by a PD controller according to the target angle (see section 2.2.2), current angle and current speed; backlash is modeled by the dead band model; and the simulation engine computes the resulted angle and torque applied; in the end, the consumed power and temperature are computed by equations (3.13) and (3.18) respectively.

3.2 Parameters Optimization

In order to obtain realistic simulation results, the parameters of the simulation have to be determined. We have two choices: either to conduct a simple experiment that measures parameters from the physic world, or to develop effective computational search procedures for finding solutions. However, measuring parameters directly from experiments requires special equipment, and some parameters of the ODE are not connected to a physical model.

Therefore, we must treat this problem as a parameter optimization: finding one or more combinations of parameter values that optimize a given performance measure. Optimization of problems of this type are generally formalized as a function optimization problem, i.e., given a function f of n arguments, $x_1, x_2, \ldots, x_n$, find the values for

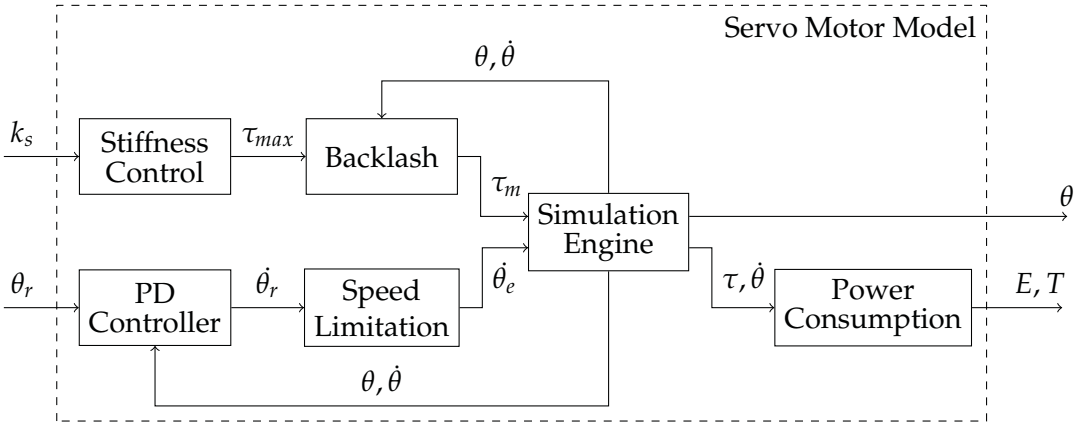


Figure 3.8: Pipeline of the servo motor simulation

$x_1, x_2, \ldots, x_n$ that maximize (or minimize) f.

These parameters usually are interdependent and setting these parameters manually is difficult or even impossible. The Evolutionary algorithm (EA) [25, 37] is a widely used optimization method, which searches a huge space of candidate hypothesis, according to the fitness function to find the best one. In this section, we optimize parameters for the simulator by EA, which uses the difference between the recorded data of a real robot and the result of the simulation as a fitness function.

3.2.1 Relevant Parameters

There are a number of parameters which influence realism and performance of the simulator. In this subsection, we describe relevant parameters of a realistic simulation for a humanoid robot and the interaction with the environment. Some of these parameters are generic for robot simulations; some of them are specific to the ODE or our models. We differentiate between the parameters that need to be optimized by the experiment and those that are known from the robot's specifications.

Known Parameters

The manufacturer has documented a set of parameters for the NAO in [2], including specification for motors, kinematics and body parts. The mass, center of mass and inertial matrix of each body part are also provided. Furthermore, the speed and torque of each motor are given. We use these values directly in our simulation. Another known parameter is the gravity coefficient which is fixed to its standard value.

Parameters for Optimization

The parameters of the joints have great impact on the performance of the simulation. In our application, there are seven parameters for each joint: the maximum velocity, the

maximum torque, the dead band of the backlash, the soft CFM, the soft ERP, and the two control factors of the PD controller. Vendor specifications of the motors regarding velocity and torque provide a good starting point for the optimization, but are definitely not directly transferable without any further optimization. There are four sets of joint parameters that have to be optimized, because the NAO robot has four different types of joints (two different types of motors combined with two different types of gears).

For the whole body motion of a humanoid robot, the interaction with environment (e.g. the ground) needs to be modeled. The contact between colliding objects can be configured by six ODE parameters: two for friction (ContactSlip and ContactMu), two for body hardness (ContactSoftCFM and ContactSoftERP), and two for bounce (ContactBounceValue and MinBounceVel). The surface of the robot and the surface of the ground need different parameters. Therefore twelve parameters need to be optimized for contact.

In total, forty parameters need to be optimized together in our application.

3.2.2 EA-based Optimization

The goal of the optimization is to find parameters for the simulator which result in a behavior of the simulated robot as close as possible to the behavior of the real robot. Because there are approximations between the simulation model and reality due to computation efficiency, the optimization is to find parameters that overall reflect close to the reality, rather than finding the real parameters (e.g. the real friction between the feet of the robot and the ground).

Evolutionary Algorithm has been widely used as an optimization technique in many fields including the RoboCup [47, 78–80]. It is inspired by the biological evolution and is working with a population of individuals. Each individual represents a point $\boldsymbol{x}$ in the n dimensional search space; the point $\boldsymbol{x}$ is encoded in the object parameters $x_1, \ldots, x_n$ of the individual and represents its genes. In terms of the EA each individual can be assigned a function value $f(\boldsymbol{x})$ representing the quality of the solution, i.e. fitness which has to be optimized. Figure 3.9 shows the process of the Evolution Algorithm. An amount of μ individuals constitutes the parent population and creates a new offspring population of the next generation with λ individuals. An offspring individual is created by recombining and mixing the real-valued genes of ρ randomly selected parents, followed by a random mutation of each object parameter such that the offspring differ in their genes from their parents. The fitness of each individual in the offspring generation will be measured and the selection operation chooses the μ best individuals to establish the new parent generation.

In the context of the EA, the parameters optimization is viewed as a problem independent paradigm for designing effective search procedures. As such, in order to be applied effectively to our particular problem, this abstract notion of an EA-based parallel adaptive search procedure must be instantiated by a series of key design decisions involving:

- deciding what an individual in the population represents,
- providing a means for computing the fitness of an individual,

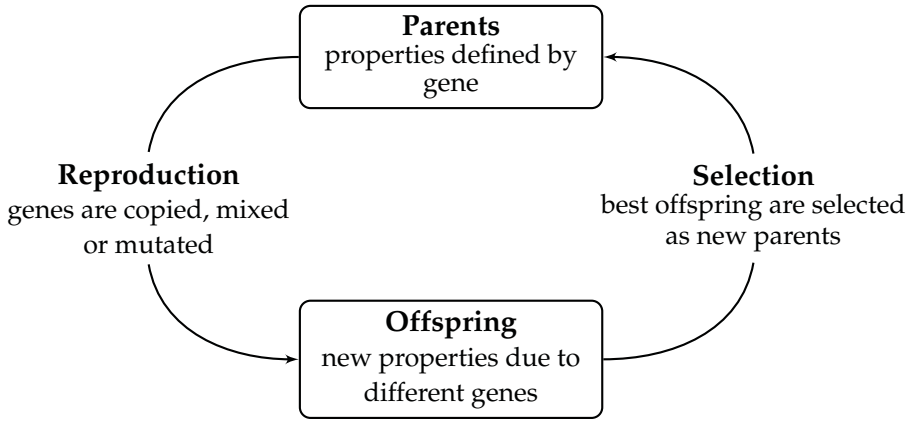


Figure 3.9: Optimization cycle of the Evolutionary Algorithm.

- deciding how children (new search points) are generated from parents (current search points),
- specifying population sizes and dynamics, and
- defining a termination criterion for stopping the evolutionary process.

3.2.3 Experiment

In our EA-based parameters optimization, the algorithm tries a set of parameters out in the simulation, and measures how closely they match the movements of the real robot as fitness.

Design and Implementation

The design and implementation of the experiment are detailed here.

Genetic Representation Since the parameters here are all real value numbers, we use the phenotypic approach, i.e. arrays of real-valued numbers, to represent chromosomes. Therefore, standard notions of mutation and recombination are easily translated into mutation operators that perturb the values of inherited parameter values, and recombination operators that select the parameter values to be inherited from multiple parents.

Fitness The fitness function is defined over the genetic representation and measures the quality of the represented solution. We use the difference between reality and simulation as fitness for the genetic algorithm. The sensor data of the robot (e.g. joint position, inertial sensor) can be used to calculate fitness. Furthermore the difference between point clouds from the Kinect and the simulation can also be used as fitness (see Figure 3.10). These criteria are not conflicting, so we can use weighted summary of all errors as fitness:

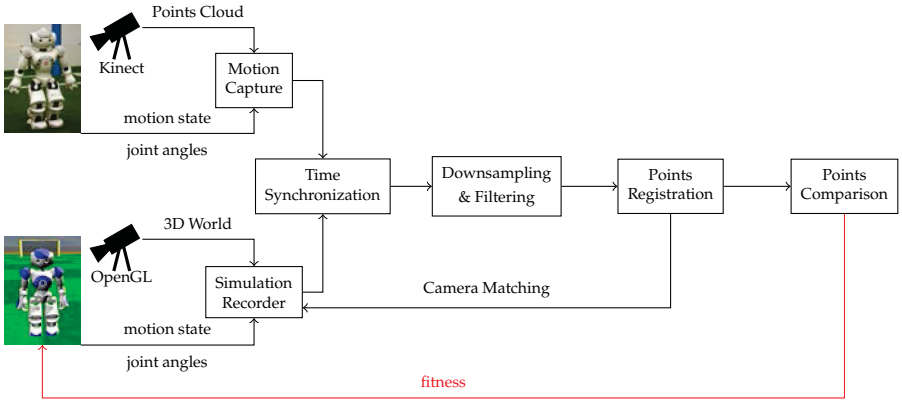


Figure 3.10: Using difference between point clouds from Kinect and simulation as fitness in Evolutionary Algorithm. The motion capture system was proposed in section 2.3.1.

$$f(\boldsymbol{s}_s, \boldsymbol{p}_s, \boldsymbol{s}_r, \boldsymbol{p}_r) = w_s \sum(\boldsymbol{s}_s - \boldsymbol{s}_r) + w_p d_S(\boldsymbol{p}_s, \boldsymbol{p}_r) \tag{3.23}$$

where $\boldsymbol{s}_s$ and $\boldsymbol{s}_r$ are sensor data from simulation and reality respectively, $\boldsymbol{p}_s$ and $\boldsymbol{p}_r$ are preprocessed points cloud data from simulation and reality respectively, w_s and w_p are weights, d_S is defined by equation (2.28).

Initialization Initially the individual solutions are seeded in areas where the original parameters of the SimSpark are, since these are likely good parameters. The population size is set to 100 in our experiment. After initialization, the EA goes into an optimization cycle (see Figure 3.9).

Selection In the stage of selection, each individual is evaluated by the fitness function and sorted by the descending fitness value, then only the μ most fit individuals are selected for next generation from this generation of individuals of size $\lambda > \mu$. The *elitist selection strategy* is also applied here, i.e. the selected parents will be put into the new population without any change. The primary advantage of elitism is rapid convergence to the optimal solution, the main disadvantage is a decrease in the probability that the global optimum will be found.

Reproduction For each new individual to be produced, a pair of "parent" individuals is selected for breeding by *roulette-wheel selection* from the pool selected previously. By producing a child using the methods of transit, crossover and mutation, a new individual is created which typically shares many of the characteristics of its parents. In particular, parametrized uniform crossover and Gaussian mutation are used. In parametrized uniform crossover, decisions about from which parent a child will inherit a parameter is

made by flipping a coin, the bias of which is controlled by a parameter p that varies from 0.0 to 0.5. Gaussian mutation is quite an effective stochastic mutation operator, in the sense that it captures an intuitive bias toward making small perturbations in parameters values.

We select new parents for each new child, and the process continues until a new population of individuals of appropriate size k is generated.

Parallel Implementation It needs a large amount of computing power to evaluate all the individuals, because for each individual (a set of parameters) the motion has to be executed in simulation and point clouds have to be generated and processed. We developed the Evolutionary Algorithm in a parallel implementation which evaluates all multi individuals in different processes at the same time. It allows us to take advantage of the server in the institute which has eight Quad-Core CPUs and 64 GB RAM.

Termination Unfortunately, unless an EA-based optimization has a specific priori of information about an optimization problem, there is no way to know when a global optimum is found. The criterion of convergence is to keep track of the best solution found so far. If that solution remains unchanged for g generations, then terminate and return to the best solution found. The optimization process can be continued by taking the last generation as the initial generation, when the result is unsatisfactory.

Results

Figure 3.11 shows the experiment result, in which the population size is 100, the elitism rate is 10%, the transmit rate, crossover rate and mutation rate are equal, i.e. 1/3.

The progress of evolution is really slow, because for each individual, (a set of parameters), the stand up motion has to be executed in simulation, and the cloud points have to be generated from the logfile of the simulation. After that we can use equation (3.23) to calculate fitness. Even when we ran the SimSpark in fast mode and ran evolution as a parallel, it still took one week to finish the optimization.

Despite the time factor, the difference between the two point clouds sequences (from the Kinect and the simulation) becomes smaller after optimization, that is, the quality of the simulation results are better. Although there is still a difference, the differences are mainly due to the noise measured from sensors.

3.3 Simulator Evaluation

In chapter 2, some of the simulation evaluation methods are introduced and used to evaluate the NAO robot in the SimSpark. In this section, some evaluation experiments were repeated after improvements were made, which were described in the last sections.

First of all, the joint servo is evaluated by step input and sinusoid input as in section 2.2.2. We get better simulation results in the new simulator (Figures 3.12 to 3.14)

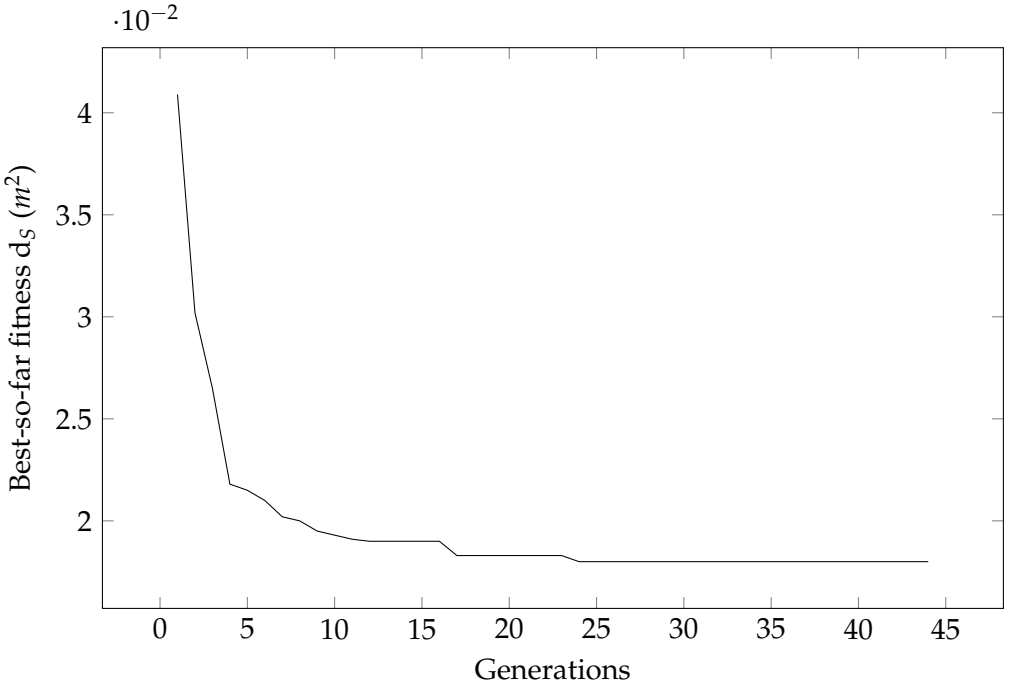


Figure 3.11: Results of the simulator's parameters optimized by the Evolutionary Algorithm. The fitness is calculated by the equation (3.23).

than in the original results (Figures 2.18 to 2.20). The simulated servo behaves the same as the real one, except that the starting moment in the sinusoid test.

The simulator is evaluated with static motions. The standing up motion designed for the real robot is repeated in the simulation without any problem (see Figure 3.15).

The evaluation result shows that the improved SimSpark behaves closer to the real robot. We hope it can be used to develop and test motions for the real NAO; otherwise we can add the failed test to the evaluation, and improve the simulator to pass this test.

3.4 Summary

This chapter describes our research for improving the realism of the SimSpark – the simulator of the RoboCup soccer Simulation League. After investigating the documentation of the NAO, we refined the robot model by using accurate information: better shape approximation, precise mass and inertia matrix. We also modeled and implemented some important sensors from the NAO, including accelerometer, RGB camera and lines in virtual vision. For the realistic actuator, we developed the servo model with backlash, power consumption and temperature protection. The experiment results show the SimSpark behaves similarly to the servo in the real NAO.

The parameters of the simulator are optimized by the Evolutionary Algorithm, where both the sensor data from the robot and the data from the Kinect are used to compare

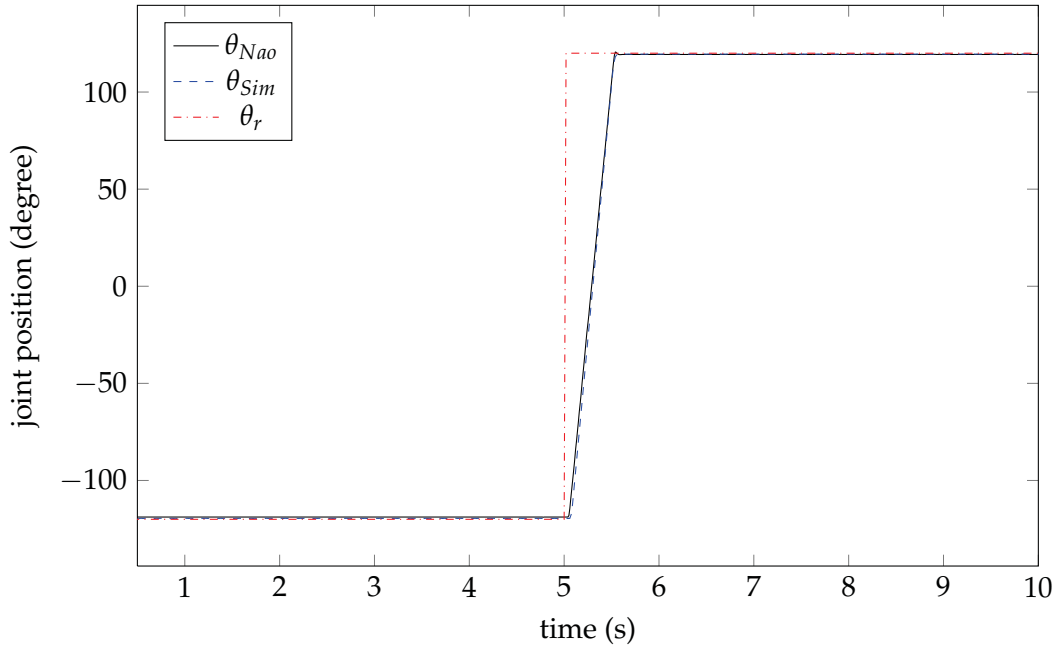


Figure 3.12: The response of HeadYaw joint to step input in the real robot and the simulation. It is the same experiment as Figure 2.18 in the improved SimSpark.

with the simulation result as fitness. After the experiment, the best result we can get from the simulation has evolved.

Moreover, we evaluated the proposed models and motion results using the same experiments proposed in section 2.3. First, the simulated joint servo performs closer to the reality (Figures 3.12 to 3.14). Second, the stand up motion could only work in reality before, and now it can also work in simulation (Figure 3.15). The experiments were repeated more than five times, therefore our model is closer to reality.

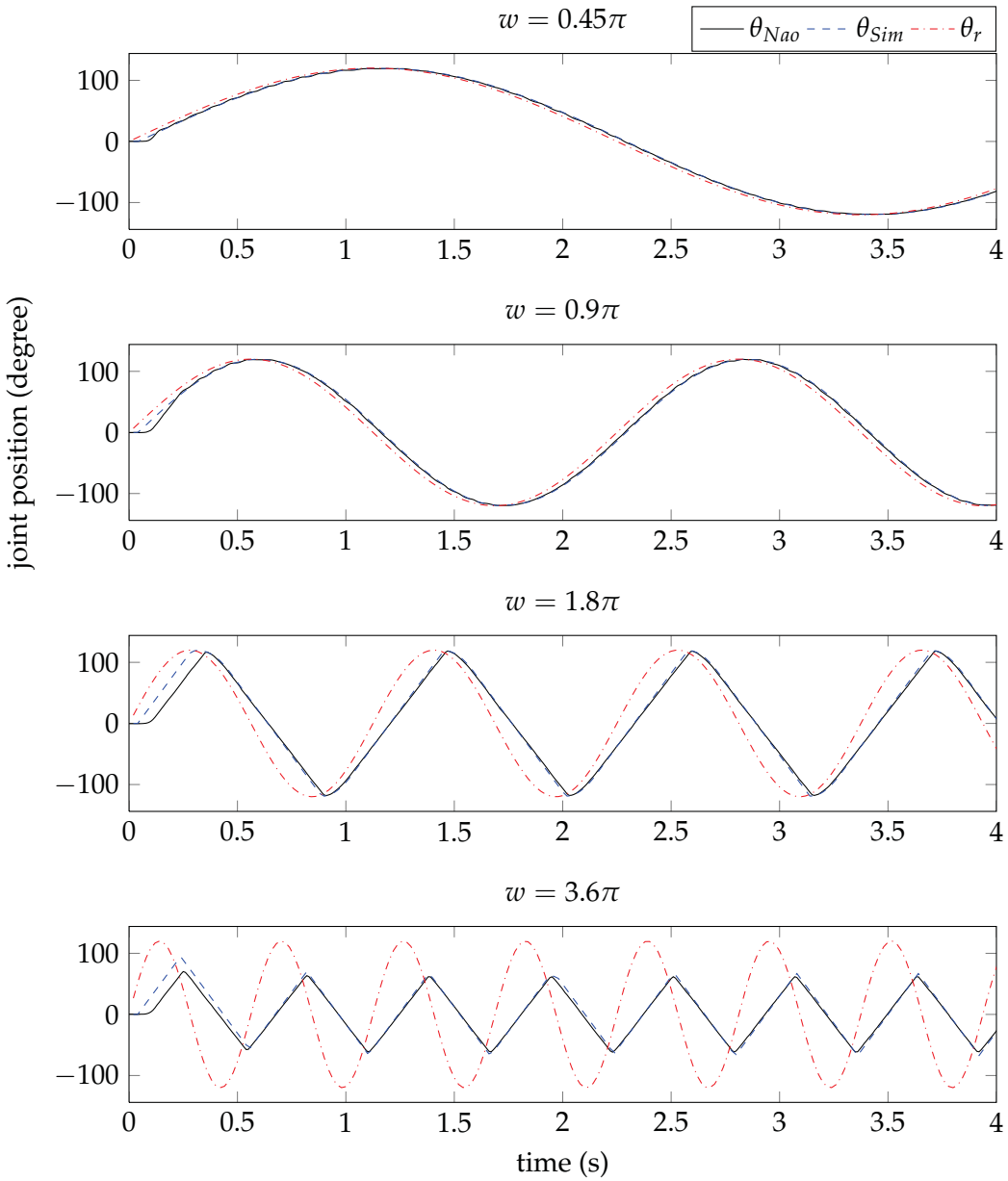


Figure 3.13: The sinusoid response of the HeadYaw joint in the real robot and the simulation with different w values in the equation (2.18). It is the same experiment as Figure 2.19 in the improved SimSpark.

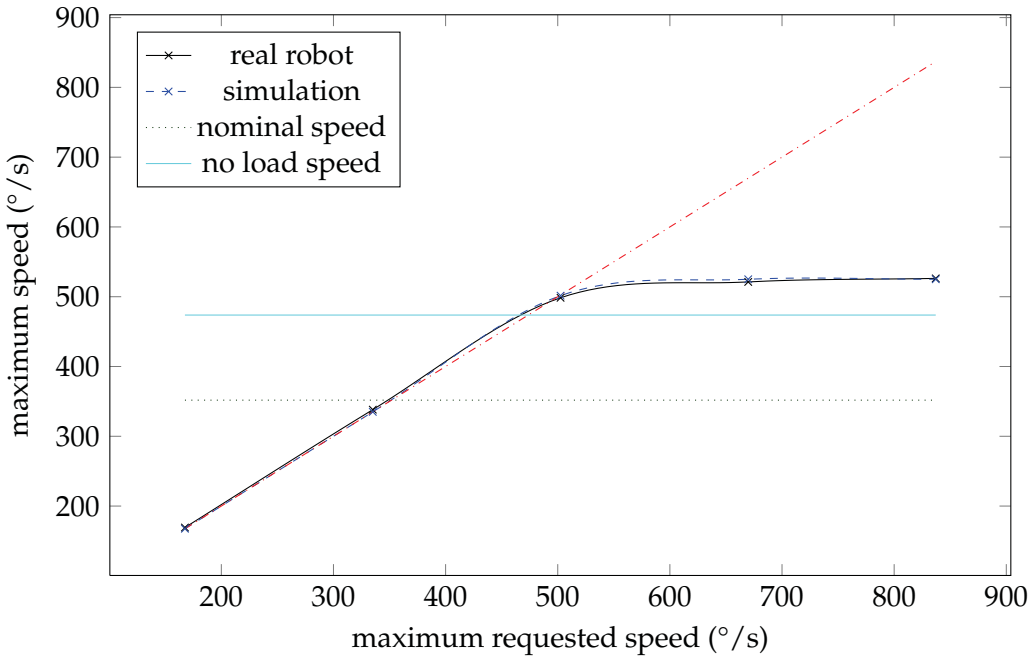


Figure 3.14: The maximum speed of the HeadYaw joint. The simulated results are better than in Figure 2.20.

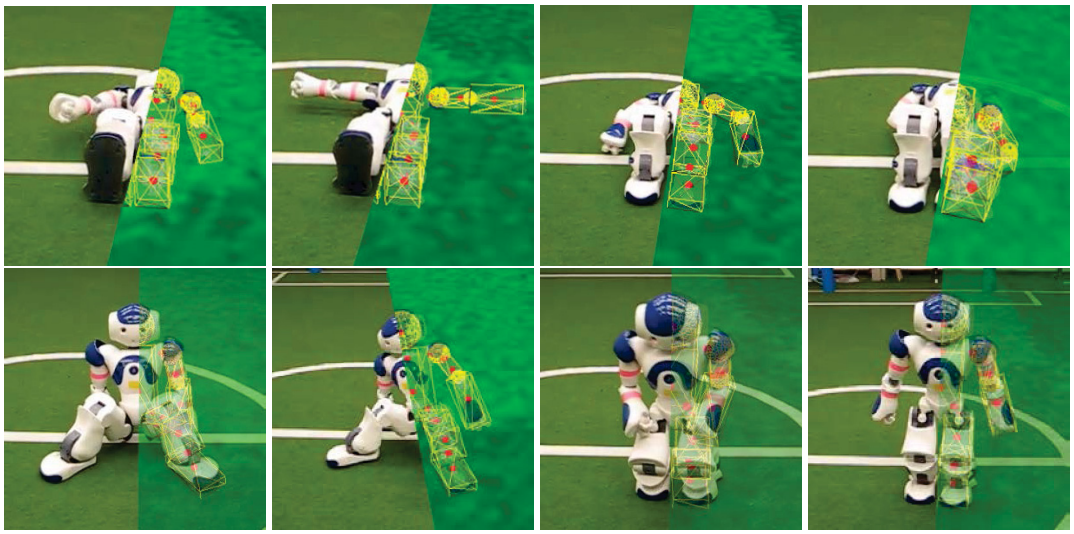


Figure 3.15: The standing up motion performed by the real NAO and the simulated NAO robot.

4 Developing Motion in the Simulation for the Real Humanoid Robot

From small beginnings come great things.

— Proverb

In order to play a soccer game, the humanoid robot must be able to perform motions as a soccer player does, including walking, kicking, standing up, etc. Motion is one of the most difficult areas in humanoid robotics research, due to the high dimensionality of a humanoid robot with its many joints, the complexity of the dynamics of the system, and the difficulty of creating an accurate model. Human like motions in a dynamic environment are still an open challenge in robotics and artificial intelligence. In the past years, many studies have been conducted on motion control of robots, and many methods have been proposed. They can generally be categorized into two groups (Figure 4.1): the *model based approach*, and the *model free approach*.

In the model based approach, a designer precisely constructs a physical model of the target system and builds a specific controller based on this model. For example, the Zero Moment Point (ZMP) is defined as the point on the ground where the sum of the moments of all the active forces equals zero [72]. If the ZMP is within the support polygon,

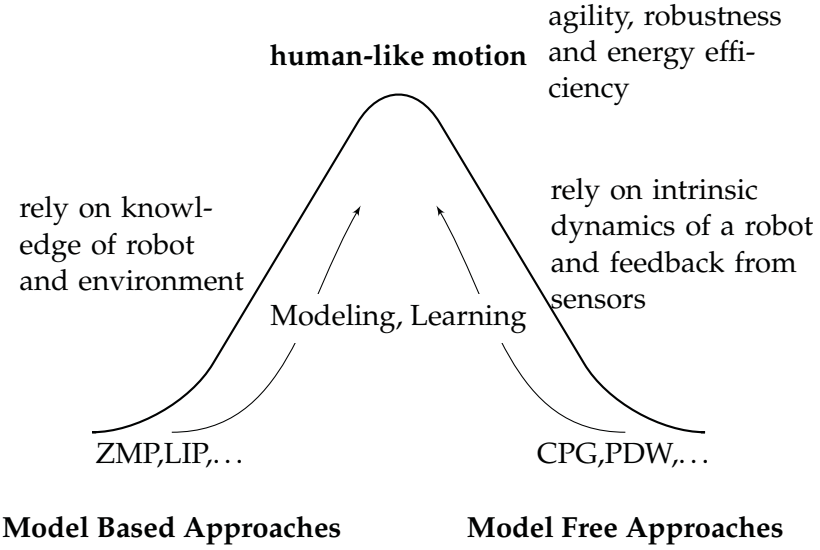


Figure 4.1: Different approaches to the human like motion for a robot.

the convex hull of all contact points between the feet and the ground, a bipedal robot is dynamically stable. To make online feedback possible, simplified models, such as the Linear Inverted Pendulum (LIP) [39, 40] method are used. When a biped robot is supporting its body on one leg, its dominant dynamics can be represented by its center of mass, which is connected by a massless telescopic leg to the supporting foot.

In the model free approach, it is more important to make use of the intrinsic dynamics of a robot or to associate the sensor information with motions. For example, in Passive Dynamic Walking (PDW) [13] - a biped robot walks down a shallow slope without any actuator. All operating forces descend from gravity. The walking motion is the result of two pendulums (the legs) swinging in their natural frequency (also known as eigenfrequency). The mechanical design of the legs, such as length, mass distribution and foot form, determines the stability of the walking motion. Another example is the Central Pattern Generator (CPG). Vertebrate animals are thought to have neural basis for locomotion in the spinal cord which are referred to as CPG [57]. CPGs are the circuits which are able to produce periodic signals in a self-contained way, i.e. without having any rhythmic input in and of themselves. These rhythmic activities can be initiated by a simple non-oscillating signal. One of the difficulties in the application of the CPG model to that of real robots is to determine the weights of the neural connections.

In NaoTH, we implemented static motions by Keyframes (cf. section 4.1), e.g. standing up, goalie jumping, etc. For dynamic motions, e.g. walking and kicking, we used simplified dynamic models of the robot to plan motions [51, 75].

Machine Learning (ML) can avoid the curse of modeling. It has attracted rapidly increasing interest in the artificial intelligence and the robotic communities in the last ten years. Although ML algorithms have been implemented and tested in simulation, ML in the real robot is difficult to experiment with due to the limited resources and time. Transferring controllers developed in simulation to real robots is difficult and it is one of the major challenges when using the ML algorithm. This is because the ML algorithms heavily explore the features of the environment, i.e. reality of the simulator.

In chapter 3, we improved the the realism of SimSpark. In this chapter, we investigated how to develop motions in simulation for the humanoid robot, especially using machine learning technology. The Evolutionary Algorithm is used to optimize and generate keyframe motions in the simulation in section 4.1. We implemented dynamic omnidirectional walking based on the simplified dynamic model in section 4.2. Furthermore, the robot learns foot trajectory and balance in simulation.

4.1 Keyframe Motion

Keyframe is one technology used to create static motions, it defines all the joint positions of the robot. A sequence of keyframes defines the motion of the robot. The joint movement between these important frames are interpolated linearly by interval time (see Figure 4.2).

Keyframe has been widely used in animations and robotics such as, dancing. The advantage of the Keyframe based method is the fact that it is easy to create a complex

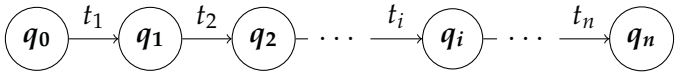


Figure 4.2: Representation of the Keyframe motion: q_i is one key frame which specifies all joint positions; t_i is the interval time between two keyframes.

motion. We can design the Keyframes by setting joint positions manually, or by teaching, i.e. moving the robot's limbs and recoding the joint positions.

The disadvantage of Keyframe motion is the lack of dynamics; thus it is slow and unnatural. It is difficult to make Keyframe Motions fast. In this section, we investigate how to optimize the speed of Keyframe Motions in the simulation and transfer them to the real robot. We also use the Evolutionary Algorithm to generate Keyframe Motion automatically.

4.1.1 Keyframe Motion Optimization

The speed of motions for playing soccer is important, therefore the Keyframe Motions for the NAO in RoboCup, such as standing up, has to be as fast as possible.

In this experiment, we optimized the interval time of the standing up motion in the SimSpark by using the Evolutionary Algorithm. This standing up motion was shown in Figure 3.15. The whole experiment setup is similar to the experiment in section 3.2. The optimization target, i.e. fitness, is the time of whole standing up motion takes if the robot can stand up successfully; otherwise the fitness is 0. Standing up from a lying back position contains 11 keyframes, and the chromosome only contains transition time, so each individual has 11 real value genes. The population size is 100, the elitism rate is 10%, the transmit rate, crossover rate and mutation rate are equal, i.e. 1/3. Although this experiment also has to execute the stand up motion in the simulation for each individual, but it doesn't need to generate cloud points as compared to the experiment in section 3.2. It is therefore faster and took about 2 days to finish.

After optimizing the speed of standing up, the resulted Keyframe Motion is evaluated in the real NAO. If the real robot fails to stand up, we record the results of motion in simulation and reality and add them to our dataset for optimization parameters for the simulator; we repeat the simulator optimization and Keyframe Motion optimization until the real robot stands up successfully with parameters optimized in the simulation.

Figure 4.3 shows the final results of the Evolutionary Algorithm: the optimized motion in the simulation can be transferred to the real robot, and repeated 3 times successfully. The optimized speed is a little faster than the speed which were turned manually.

This means we can create a Keyframe Motion manually at very slow speed, and then optimize the speed in the simulation. However, this method can not improve the speed too much, because the slow Keyframe Motion are created with a static stabilization criterion. The optimization can not take advantage of the dynamics of the robot. This reminds us that the time intervals are combined together with key frames, so they need to evolve together with key frames.

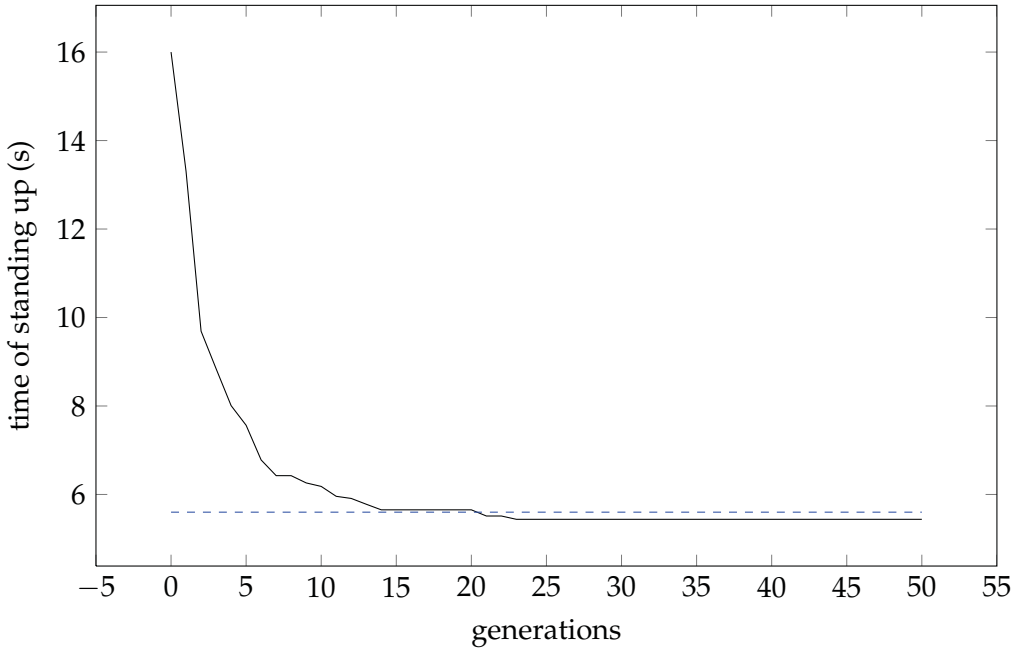


Figure 4.3: The speed of standing up from a lying back position is optimized by the Evolutionary Algorithm. The dashed line indicates the time of the manually turned values.

4.1.2 Keyframe Motion Auto-generation

The keyframes created by setting values manually or even by teaching, is usually unnatural. Because the robot has different kinematics and dynamics from a human, the keyframes of a human in motion are not suitable for robots. In this section, we investigate to generate keyframe motion by exploring in simulation.

Our method is to use the Evolutionary Algorithm to explore motion space in the simulation. The general process is the same as illustrated in Figure 3.10, but the genetic representation of the solution is different: a list as illustrated in Figure 4.2 is used. In the list, each node contains joint positions $\boldsymbol{q_i}$ and transferring time t_i, $\boldsymbol{q_o}$ and $\boldsymbol{q_F}$ are given in the target motion.

The genetic operations are also different: the crossover combines two different parts of the list from the parents (see Figure 4.4). In the mutation, one node of the list is randomly selected, and replaced by a new randomly generated node (see Figure 4.5).

The design of the fitness function is not straightforward. For the standing up motion, the height of the center of mass is an important factor. EA can not converge when only standing up successfully or not is used as fitness, it needs the intermediate fitness to converge faster. We include the height of the center of mass in each simulation cycle into

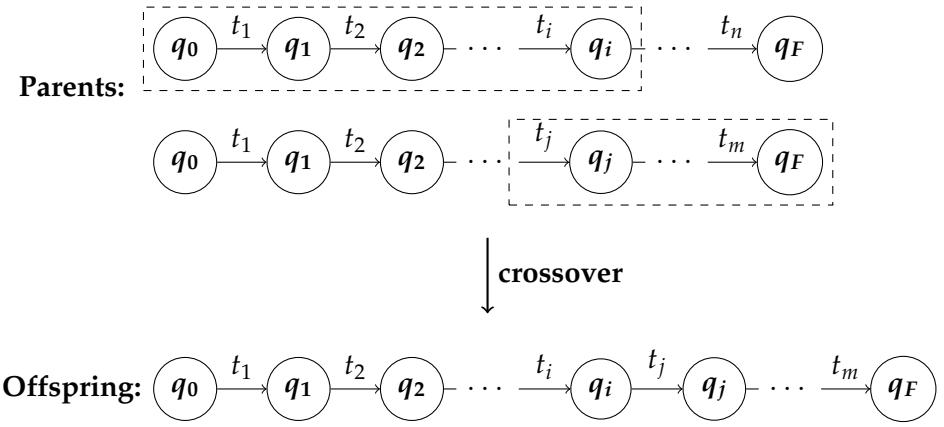


Figure 4.4: The crossover operation with two different parents: two options from the list from different parents are combined as a new individual.

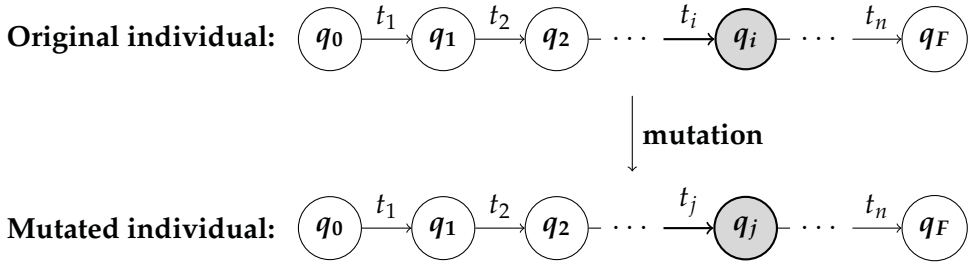


Figure 4.5: The mutation operation: one node from the list is replaced by a newly generated node.

the fitness:

$$f(t_1, h_1, t_2, h_2, \ldots, t_n, h_n) = \frac{\sum_{i=1}^{n} (h_i \sum_{j=1}^{i} t_j)}{H \sum_{i=1}^{n} \sum_{j=1}^{i} t_j} \tag{4.1}$$

where h_i is the height of CoM at time t_i, and H is the target height after standing up successfully, so the value of this function is normalized in range $[0, 1)$. We weighted the value of CoM according to time because the final height is more important than in earlier steps, i.e. standing up successfully is more important than standing up faster but the motion failed in the end.

We set the parameters of EA as follows: the population size is 500, elitism rate is 2%, the transmit rate, crossover rate and mutation rate are equal, i.e. $1/3$. Due to the higher dimension of exploring space, it is much slower than the experiment in section 4.1.1.

Figure 4.6 illustrates one successfully auto-generated keyframe motion after 3 weeks; and the real robot successfully replayed this motion. However, comparing this generated motion with the motion designed manually, it is slower. The speed of generated motion can be optimized by the approach in section 4.1.1, this could be the improvement in the

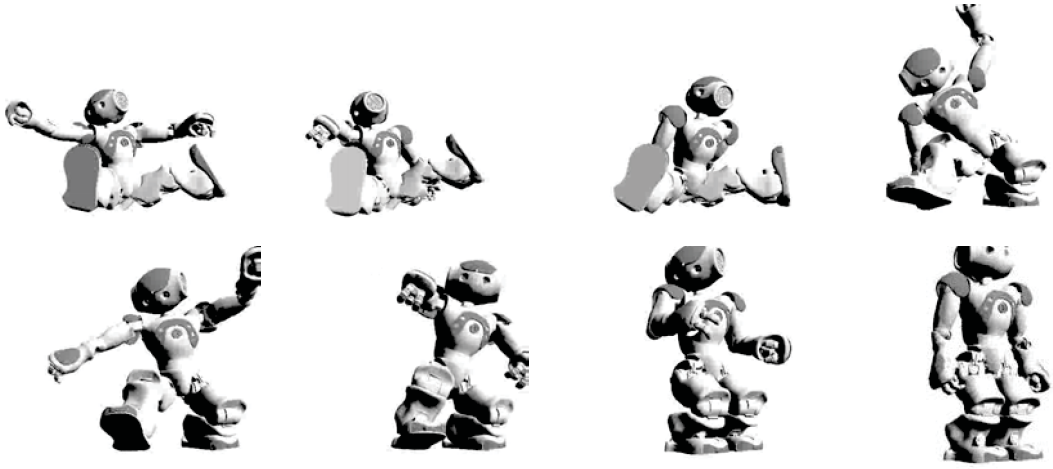


Figure 4.6: The standing up motion from a lying back position generated by the evolution automatically.

future.

In this section, we investigated how to optimize the keyframe motion and auto-generate the keyframe motion in simulation, and then apply the results in the real robot. The experiment results show that, the pre designed keyframe motion can be optimized in simulation and used in the real robot, however the improvement is limited. Auto-generating a keyframe motion such as standing up by using the EA is also possible, however we have a really high dimension of space to explore, and therefore it took a lot of time.

4.2 Dynamic Biped Walking

Despite a large amount of literature on this topic, human like walking for humanoid robots is still an open challenge. It is a hard problem due to the high dimensionality of a humanoid robot with many joints, the complexity of the dynamics of the system, and the difficulty of creating an accurate model. In the past 30 years, many studies have been conducted on biped locomotion control, and many biped locomotion methods have been proposed [5, 13, 17, 32, 39, 40, 57, 72, 77].

In order to play soccer, the robot should be able to change destination dynamically and walk in any direction (i.e. omnidirectional). However, most of the past studies concentrated on the periodic and stable biped locomotion. They are not suitable to the dynamic and adversarial environment such as with RoboCup. Recently, RoboCup teams in the Standard Platform League have achieved some good results [17, 32, 77]. They are all model based approaches, and all of them use the *Linear Inverted Pendulum* (LIP) [39] model to control the *Zero Moment Point* (ZMP) [72] of the robot.

We investigate both model based and model free approaches for this challenging prob-

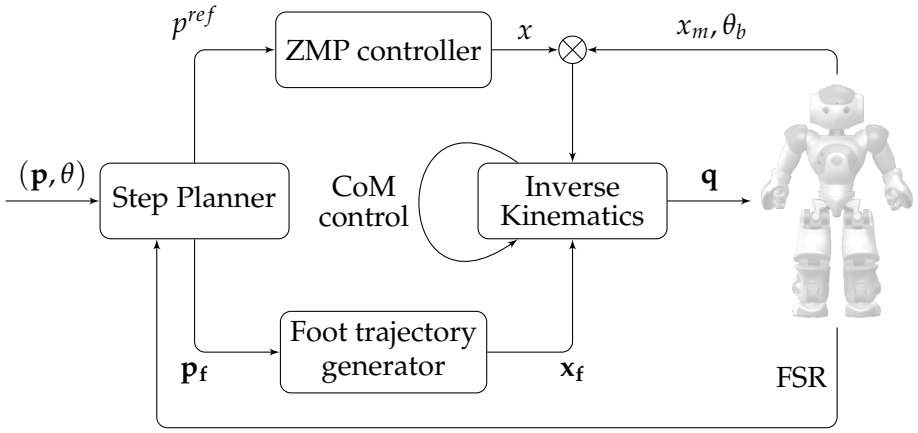


Figure 4.7: The architecture of our walking engine. Red lines indicate the sensor feedback.

lem. We used the ZMP model as stability criterion and the LIP model to approximate the dynamics of the robot. We used evolution to optimize the parameters and Machine Learning to learn foot trajectory and to maintain balance. After the development in the simulation, we applied and tested walking in the real robot.

4.2.1 Walking Engine

The walking engine is a sub-module in the motion module of the NaoTH. It accepts the *walk request* from the cognition module. The walk request gives the destination of the walk, which includes the 2D position (x_r, y_r) and rotation θ_r. All of them are in the robot's local coordinate system. It is useful to stop the walk with the given foot position as the destination, e.g. going to the ball and preparing to kick. Therefore, the walk request contains an additional flag c to indicate the origin of the destination: left foot, right foot or chest. The task of the walk is defined by the walk request (x_r, y_r, θ_r, c).

The walking engine has to move the feet to the destination given by the walk request, and keep balance at the same time. We divided the whole process into 5 stages or tasks as shown in Figure 4.7. Firstly, the step planner plans the foot's step $\boldsymbol{p}_f$ and ZMP p^{ref}; then foot trajectory generator generates foot trajectory $\boldsymbol{x}_f$ in one step, and the ZMP controller calculates the target's center of mass x according to p^{ref}; at the end, the joint angles $\boldsymbol{q}$ are determined according to x and $\boldsymbol{x}_f$ by solving the inverse kinematics. Additionally, we used the sensors data as feedback to stabilize the robot in different stages. The next paragraphs give details of each component.

Step Planner

In our implementation, the cognition module responds to plan the walking path in global coordinate system. The task of the step planner is determining where to put the moving

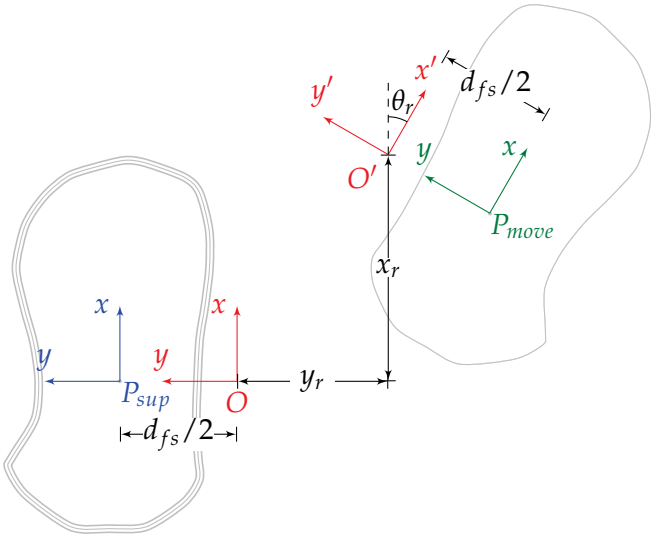


Figure 4.8: The coordinate system of the walking engine: d_{fs} is the distance between the two feet while the robot is standing in the initial position, O is the current pose of the robot, and O' is the pose of the robot after a step (x_r, y_r, θ_r). P_{sup} and P_{move} are poses of the support foot and the moving foot in this step separately.

foot in the next step according to the walk request (x_r, y_r, θ_r, c).

First we have to choose the local coordinate system. It is not a good idea to choose the torso to represent the robot's pose, because the center of mass is controlled for the ZMP criterion in order to maintain balance. The pose of the torso is changing all the time during walking and it can not be determined by the step planner. We chose the support foot as the origin of coordinate system (see Figure 4.8). This makes sure the planned step will be achieved. As mentioned before, requesting the foot pose in next step is necessary for going to the ball and preparing to kick. In this coordinate system, only a fixed transformation is required according to which foot is requested.

The rest of the job of the step planner is making sure that the resulted moving foot pose can be reached in the next step. However it is not as easy as it seems to due to the kinematics and dynamic constraints, and the self collision of the robot. A range limitation is applied to (x_r, y_r, θ_r) for the kinematics constraints and the self collision. An ellipsoid is used to approximate the limitation in 3D at the moment. For the dynamic constraints, we used feedback from the FSR and the inertial sensor to adjust the moving foot.

Foot trajectory

Because the beginning and ending poses are determined by the step planner, the motion of foot in this step, i.e. the foot trajectory, has to be interpolated. For movement of the robot's foot, an interpolation function was selected with the following three criteria:

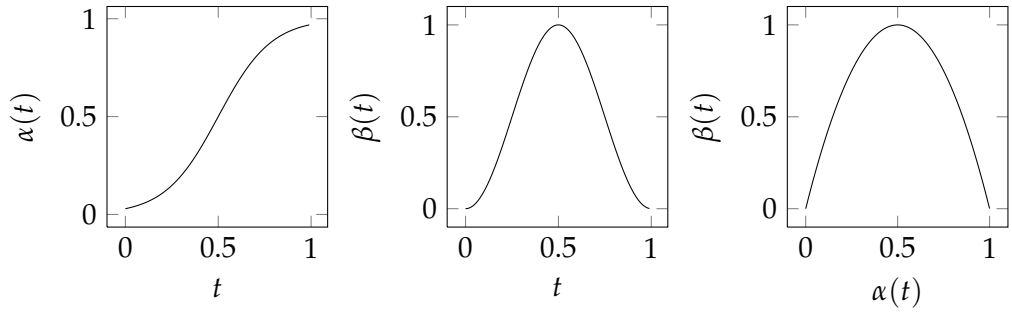


Figure 4.9: The foot trajectory during normal walking (see equations (4.3) and (4.5)) when $\lambda = 7$. $\alpha(t)$ and $\beta(t)$ are the foot trajectory in horizontal and vertical planes respectively.

- the horizontal velocity should be zero while the foot is in contact with the ground;
- keeping the loft foot away from the ground as much as possible;
- the trajectory of the foot should be smooth, i.e. velocity is continuous.

The *NaoTH-2011* uses the following equations to generate foot trajectory:

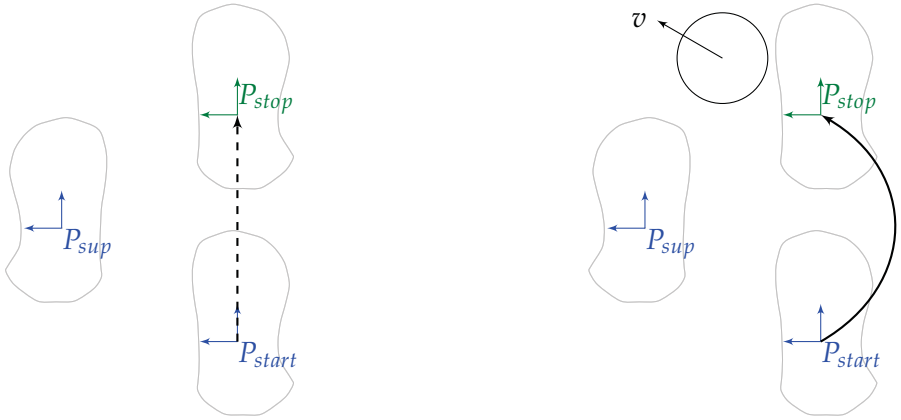
$$\alpha(t) = 1/(1 + \lambda e^{-(t-0.5)}) \tag{4.2}$$
$$s(t) = (1 - \alpha(t))S_0 + \alpha(t)S_1 \tag{4.3}$$
$$\beta(t) = \cos((t - 0.5)\pi) \tag{4.4}$$
$$h(t) = \beta(t)H \tag{4.5}$$

where, t indicates the normalized time cycles from 0 to 1; $s(t)$ and $h(t)$ are the horizontal and vertical foot position during the step t; λ is the foot curve factor, it is determined by experiments; S_0, S_1 and H are given values: S_0 and S_1 are the horizontal position at the beginning and ending; and H is the maximum foot height. The rotation of the moving foot is also interpolated by equation (4.3). Figure 4.9 shows the resulting trajectory in the horizontal and vertical planes.

For the trajectory of a special step control, the trajectory in the horizontal plane $\alpha(t)$ doesn't use equation (4.3), instead, it is interpolated by cubic splines with the start and end positions of moving foot. This allows the robot to kick the ball during walking (see Figure 4.10).

Zero Moment Point Control

The *Zero Moment Point* is defined as the point on the ground where the sum of the moments of all the active forces equals zero, i.e. the distributed floor reaction force can be replaced by a single force that acts on ZMP. If the ZMP is within the support polygon – the convex hull of all contact points between the feet and the ground, a bipedal robot



(a) Normal walking step.

(b) Kicking the ball during walking.

Figure 4.10: An example of the special foot trajectory: kicking the ball during walking.

is dynamically stable [72]. According to this theory, we have to control the ZMP inside the support foot during walking, therefore the center of the support foot is chosen as the ZMP reference.

Due to the high dimensionality of a humanoid robot with many joints, it is difficult to calculate the ZMP, control the ZMP, and use sensor data as feedback. But if we simplify the multi masses model to one mass model, the relationship between the center of mass and the ZMP is much simpler:

$$p_x = x - \frac{z\ddot{x}}{\ddot{z} + g} \tag{4.6}$$

$$p_y = y - \frac{z\ddot{y}}{\ddot{z} + g} \tag{4.7}$$

where, (x, y, z) is the position of CoM, z is in the vertical direction, g is the acceleration of gravity, and (p_x, p_y) is the ZMP.

Kajita [39] proposed the *Linear Inverted Pendulum* model, which assume the height of CoM is fixed during walking (see Figure 4.11).

It means $\ddot{z} = 0$, therefore the height of CoM is constant z_c, and

$$p_x = x - \frac{z_c\ddot{x}}{g} \tag{4.8}$$

$$p_y = y - \frac{z_c\ddot{y}}{g} \tag{4.9}$$

In order to follow the target of the ZMP trajectory by moving CoM smoothly, Kajita

Figure 4.11: The biped robot is approximated as a 3D Linear Inverted Pendulum: the support point is a ball joint which can rotate freely, the center of mass moves in a fixed plane.

[40] proposed to minimize the object function:

$$J = \sum_{i=l}^{\infty} \{Q(p_i^{ref} - p_i)^2 + R\ddot{x}^2\} \tag{4.10}$$

where Q, R are the weight of the objects function. This problem can be solved by the optimal control theory with a preview controller:

$$\ddot{x}_k = \underbrace{-K_x x_k}_{\text{state feedback}} - K_I \underbrace{\sum_{i=0}^{k}(p_k - p_k^{ref})}_{\text{Accumulated ZMP error}} - \underbrace{[f_1, f_2, \ldots, f_N]}_{\text{Preview gain}} \underbrace{\begin{bmatrix} p_{k+1}^{ref} \\ p_{k+2}^{ref} \\ \vdots \\ p_{k+N}^{ref} \end{bmatrix}}_{\text{future ZMP}} \tag{4.11}$$

where p_k^{ref} is the target ZMP at time k, p_k is the resulting ZMP at time k, N is the number of preview cycles, it is 50 when the preview time is 0.5 second for NAO, and $K_x, K_I, [f_1, f_2, \ldots, f_N]$ are parameters of the preview controller, which can be calculated by solving a Riccati equation.

In each control cycle, a new target ZMP is added into the list of future ZMP and remove the oldest one from the list, the acceleration of CoM $\ddot{x}_k$ is calculated by equation (4.11), and finally the CoM x_k is calculated by x_{k-1} and $\ddot{x}_k$. Figure 4.12 shows the results of the preview controller: the error of the resulting ZMP is acceptable; and the movement of CoM is smooth. The only drawback of the preview control is that a number of the target ZMP has to be pre-determined and this it makes the robot unable to react immediately.

Note that for the ZMP in the longitudinal direction, we have exactly the same con-

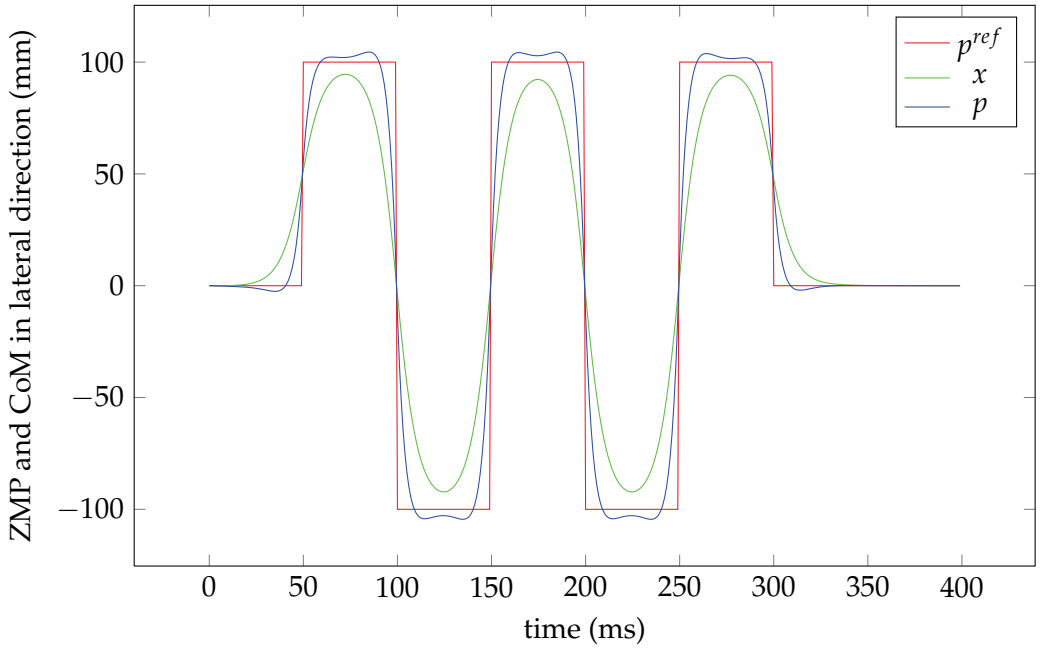


Figure 4.12: The comparison between the target ZMP (p^{ref}) and the resulting ZMP (p) of a preview controller where x is the CoM.

troller. We set the target ZMP always in the center of the support foot, the foot step and trajectory of CoM are determined, therefore the walk motion is calculated. Figures 4.13 and 4.14 show examples of the omnidirectional walk in the real NAO.

Inverse Kinematics and CoM control

So far, the center of mass $\boldsymbol{x}$ and poses of the two feet $\mathbf{x_f}$ in each control cycle are determined. The robot has to move its joints to satisfy the relation between $\boldsymbol{x}$ and $\mathbf{x_f}$. Since the robot can control poses for its torso and feet in the Cartesian coordinate system by using Inverse Kinematics (IK) technology, we use an online minimization algorithm to optimize the pose of the torso by giving poses of the feet and the center of mass. The target of the minimization is the error between the requested and the resulting CoM. For reducing the computation, the results of the last control cycle is used as the initial value of the minimization.

Solving the Inverse Kinematics problem for the NAO is not straightforward because of the special design of the hip yaw joint: they are rotated by 45 degrees and driven by a single servo motor. So there is one degree of freedom missing. The robot may be unable to reach given poses. We solve this problem by letting the algorithm find the proper yaw rotation of the torso, which is not determined by previous processes. We calculate the yaw rotation of the torso by a minimization algorithm for the difference between the resulting two hip yaw joints. Figure 4.15 describes the process, where solveOneLegIK

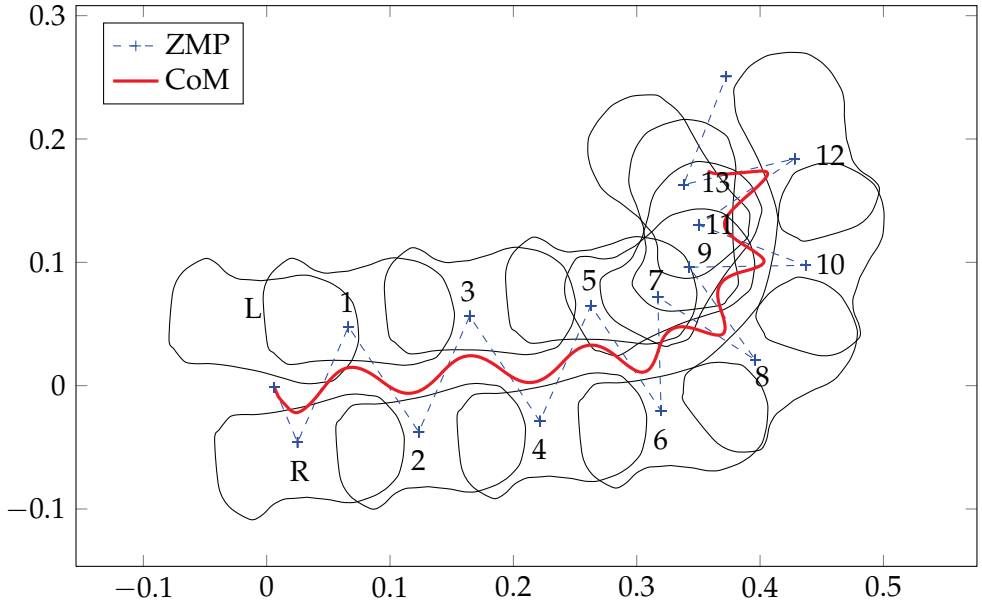


Figure 4.13: An example of the omnidirectional walk when the target ZMP is in the center of support foot. The robot is walking forward and turning at the same time.

calculates joint angles analytically or numerically by using the classic IK technology [40]; and we use the analytical solution in the NAO for performance reasons.

By using the CoM control, the robot can calculate all the joint angles of the leg from the given CoM and poses of the feet. In the open loop walk, the resulted joint angles are sent to the robot servo for executing motions.

Modification of the Walking Trajectory based on Sensor Feedback

It might be possible to find parameters that allow the robot to walk slowly, but it is obvious that the open loop walk is not enough for fast walking. Because the LIP model is simply not precise enough to represent the dynamics of the robot, the ground is not perfectly even and the joint may fail to reach the requested angle. To solve this problem, we use the sensor feedback; modifying the walking trajectory based on the sensor feedback. NAO has several sensors which can be used for this task, e.g. the force sensor in the feet, and the inertial sensor in the torso. In the *NaoTH-2011*, we use the following simple approaches:

- modifying the foot step size by the CoM: decreasing the step size when the difference between the observed CoM and the expected CoM becomes bigger;
- modifying the step time with the force sensor in the feet: increasing the step time when the foot does not touch the ground as expected;
- modifying the roll and pitch of the torso according to the inertial sensor data.

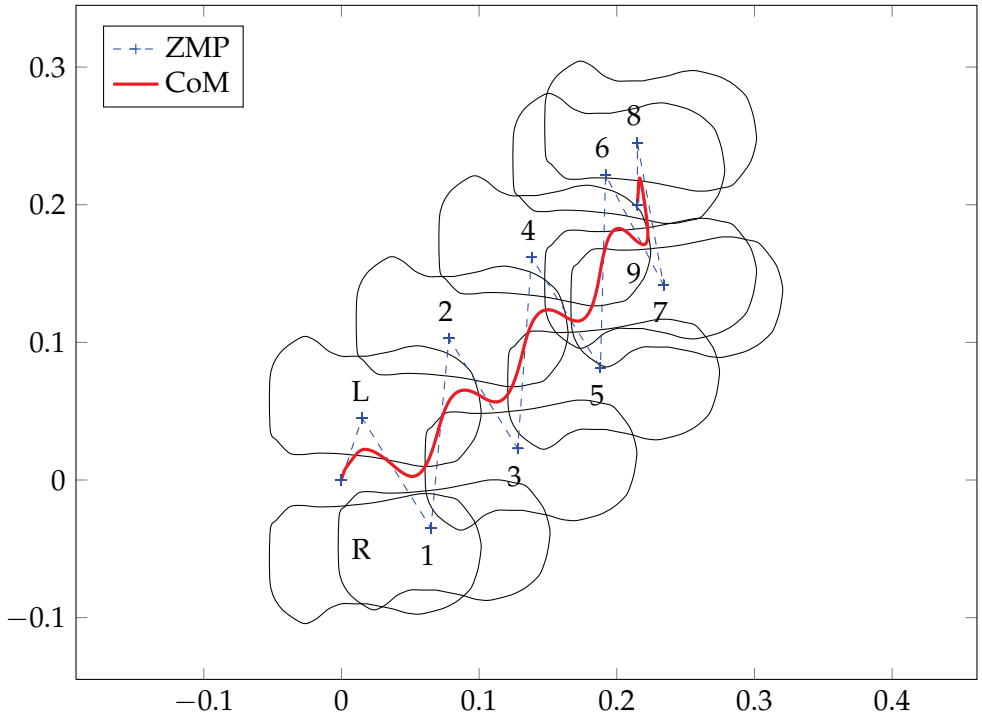


Figure 4.14: Another example of the omnidirectional walk when the target ZMP is in the center of support foot. The robot is walking forward and side ways at the same time.

After the PID controllers are used in the stabilizers of the walking engine, the robot is able to deal with any disturbance and walk instability for long time.

4.2.2 Optimize Parameters

We use the Evolutionary Algorithm as well to optimize the 18 parameters of our walking engine in the simulation. The fitness is calculated by how far the robot can walk in 30 seconds. The population size is 100, the elitism rate is 10%, the transmit rate, crossover rate and mutation rate are equal, i.e. 1/3. For comparison, we repeated the evolution with the different simulation models: 1) the original model in the SimSpark, 2) the model developed in chapter 3 without the over heating protection, and 3) the model developed in chapter 3 with the over heating protection (see Figure 4.16) for results. The resulting parameters are evaluated in the real robot.

In experiment 1, the robot can walk much faster with the original model than the new model. it is one of the fastest walkers in the Simulation League, but it is useless for the real robot, the robot can not even make one step with these parameters. In experiment 2, the resulted parameters can be applied in the real robot, and the robot can walk, but it is unnatural and unstable. This is because the robot walks at a low height. After considering

Define: solveOneLegIK($\boldsymbol{T}, \boldsymbol{F}$) solves Inverse Kinematics for one leg by given pose of torso $\boldsymbol{T}$ and foot $\boldsymbol{F}$, returns joint angles
Define: yaw($\boldsymbol{P}$) denotes the yaw rotation of pose $\boldsymbol{P}$
Define: $\boldsymbol{L}$ the target pose of left foot
Define: $\boldsymbol{R}$ the target pose of right foot
Define: $\boldsymbol{T}$ the target pose of torso, which yaw rotation is not determined
Define: λ the minimization steps which in range of $(0, 1)$, optimized by experiments
Define: MAX_HIP_ERROR the threshold of termination

```
yaw(T) ← 0.5(yaw(L) + yaw(R));
while True do
    q_l ← solveOneLegIK(T, L);
    q_r ← solveOneLegIK(T, R);
    e ← q_l[0] − q_r[0] ;            /* 0 is the index of hip yaw joint */
    if e < MAX_HIP_ERROR then
        break;
    yaw(T) = yaw(T) + λe;
return q_l, q_r
```

Figure 4.15: Pseudo code of the Inverse Kinematics for two legs of the NAO.

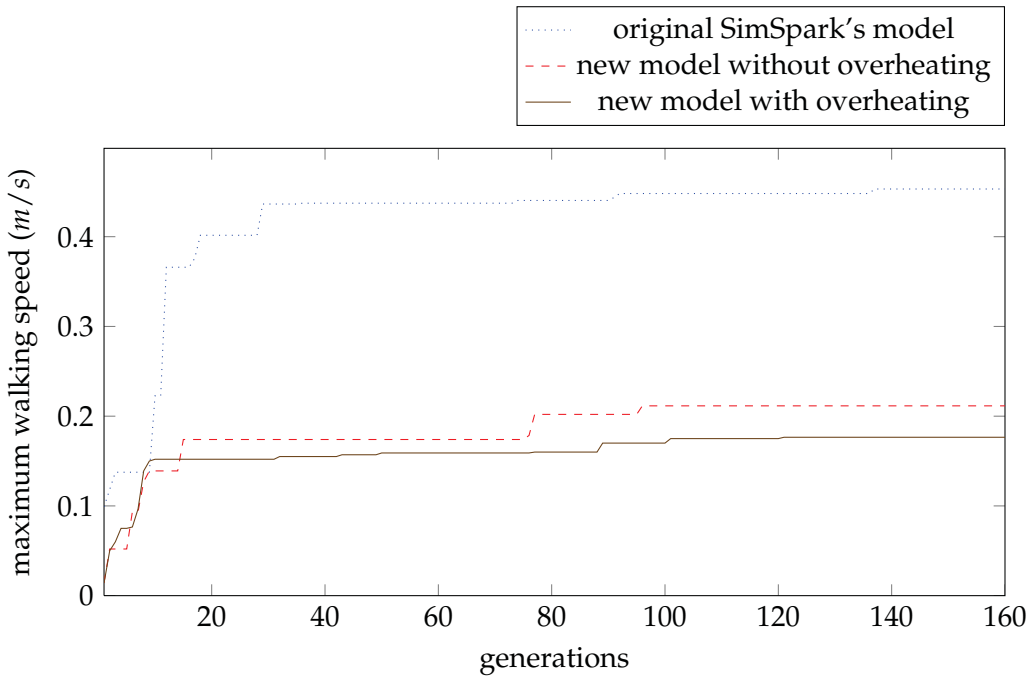


Figure 4.16: The walking speed is optimized by genetic algorithm with three different models.

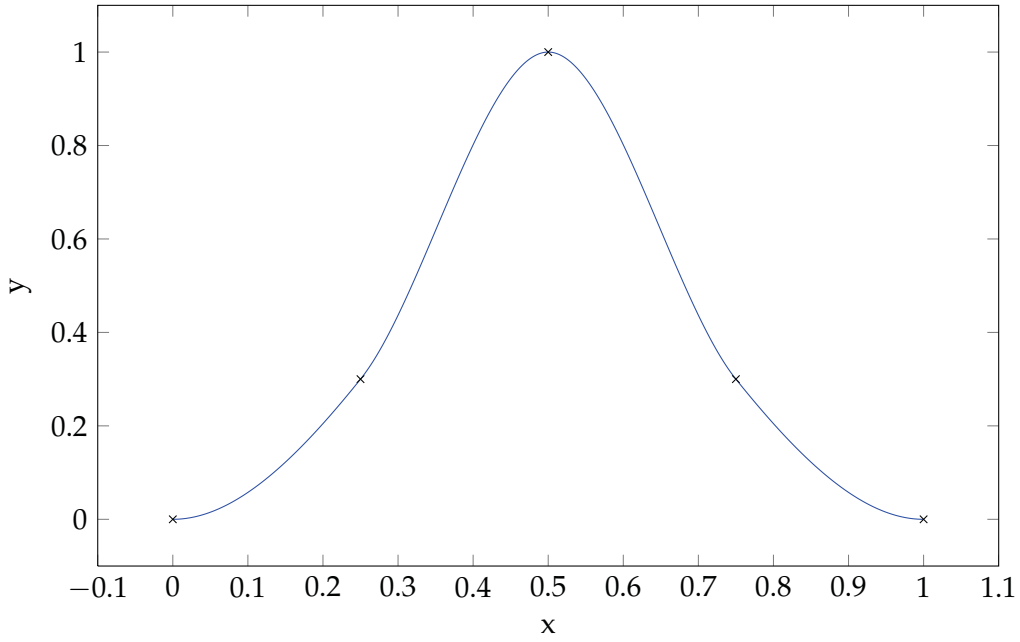


Figure 4.17: A cubic spline composed of four polynomial segments.

the power consuming in experiment 3, the robot walks slower but much more naturally and is more stable.

4.2.3 Learned Foot Trajectory

In section 4.2.1, we predefined the foot trajectory during one step. The pre-definition relies on the experience of the designer and there is no guarantee it will be optimal for robot. In this section, we select the foot trajectory with experiments. First, we represent the foot trajectory with knots (key points) using an interpolation function instead of a pre-defined function, then the foot trajectory is learned through evolution. The main task is to choose the interpolation function and design the evolution experiment.

In computer graphics splines are popular curves because of the simplicity of their construction, their ease and accuracy of evaluation, and their capacity to approximate complex shapes through curve fitting and interactive curve design. In mathematics, a spline is a sufficiently smooth piecewise-polynomial function (see Figure 4.17). Spline interpolation is often preferred to polynomial interpolation because it yields similar results, even when using low-degree polynomials, while avoiding Runge's phenomenon for higher degrees.

Considering the velocity of the foot trajectory has to be continuous. We use the cubic spline, i.e. a third-degree spline for each polynomial. In particular, the cubic Hermite spline is used in our implementation. A cubic Hermite spline consists of two control points p_k, p_{k+1} and two control tangents m_k, m_{k+1} for each polynomial in interval $[t_k, t_{k+1}]$:

$$\mathrm{p}(t) = \mathrm{h}_{00}(x)p_k + \mathrm{h}_{10}(x)(t_{k+1} - t_k)m_k + \mathrm{h}_{01}(x)p_{k+1} + \mathrm{h}_{11}(x)(t_{k+1} - t_k)m_{k+1} \tag{4.12}$$

with $x = (t - t_k)/(t_{k+1} - t_k)$ and h refer to the basis functions, defined below:

$$\mathrm{h}_{00}(x) = 2x^3 - 3x^2 + 1 \tag{4.13}$$
$$\mathrm{h}_{10}(x) = x^3 - 2x^2 + 1 \tag{4.14}$$
$$\mathrm{h}_{01}(x) = -2x^3 + 3x^2 \tag{4.15}$$
$$\mathrm{h}_{11}(x) = t^3 - t^2 \tag{4.16}$$

The foot trajectory in the horizontal and vertical plane, i.e. the $\alpha(t)$ in the equation (4.3) and $\beta(t)$ in equation (4.5), are defined by two cubic Hermite splines with 5 separate knots. For tangents m_k, we use the Catmull-Rom formula:

$$m_k = \begin{cases} \frac{p_{k+1} - p_{k-1}}{t_{k+1} - t_{k-1}} & 0 < k < 5; \\ 0 & \text{otherwise.} \end{cases} \tag{4.17}$$

The beginning and the ending knots are known, e.g. (0, 0) for beginning and (1, 0) for ending (see section 4.2.1 for details), therefore in total only six values need to be learned with evolution.

The power consumed by walking is token as optimization target, because more natural motion needs less energy. At the same time, the robot also has to walk stably the given distance, otherwise the fitness is 0. The parameters of EA are chosen as follows: The population size is 100, the elitism rate is 10%, the transmit rate, crossover rate and mutation rate are equal, i.e. 1/3.

Figure 4.18 shows the least power consumption in the evolution. The resulting foot trajectory consumes less energy than the predefined one in section 4.2.1. Figure 4.19 shows the resulting foot trajectory, which are unsymmetrical, but less power consuming, and the walking motion looks more like a human walk. Because we have fixed the step size and the step duration here, we get a more energy efficient foot trajectory.

4.2.4 Learning to Balance

In section 4.2.1, we use three feedback methods to stabilize the robot. However, designing the feedback controller relies on the experience of experts. Furthermore, it is not easy to integrate data from different sensors into one controller, and so there is a priority problem in the multi-controllers system which are based on different sensors separately. In this section, we take advantage of the simulator using the Reinforcement Learning (RL) instead of the PID controllers to keep the balance during walking, and try to solve the aforementioned problems.

In the RL, the computer is simply given a goal to achieve, and learns how to achieve that goal by trial-and-error interactions with its environment. RL dates back to the early

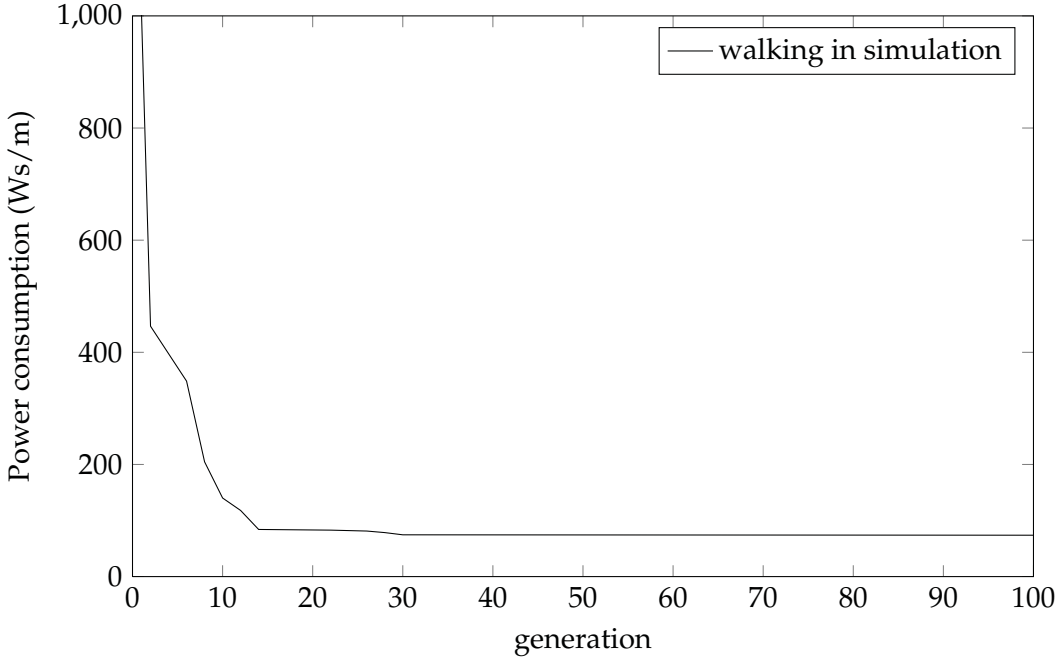


Figure 4.18: The power consumption is optimized by the learning foot trajectory; the predefined trajectory is the equations (4.3) and (4.5).

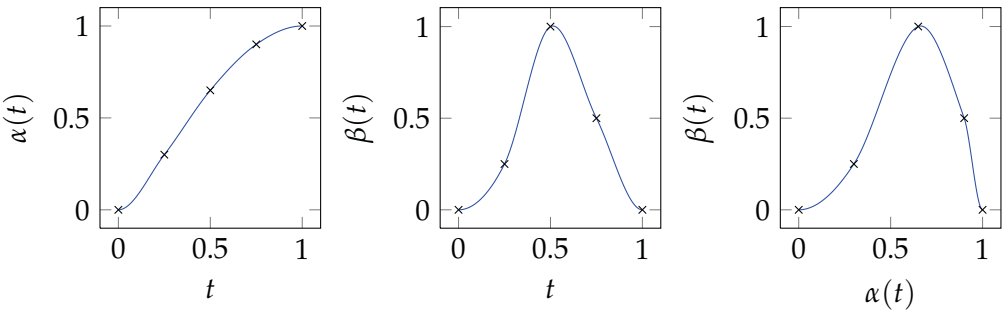


Figure 4.19: The foot trajectory learned in the simulation.

days of cybernetics and work in statistics, psychology, neuroscience, and computer science. In the last ten years, it has attracted rapidly increasing interest in the machine learning and artificial intelligence communities. Its promise is beguiling. It is a way of programming agents using reward and punishment without needing to specify how the task is to be achieved. RL allows, at least in principle, to bypass the problems of building an explicit model of the behavior to be synthesized.

Q-learning is a reinforcement learning technique that works by learning an action-value function that gives the expected utility of taking a given action in a given state and following a fixed policy thereafter. One of the strengths of Q-learning is that it is able

to compare the expected utility of the available actions without requiring a model of the environment.

It is the job of the RL system designer to define the state $\mathcal{S}$ and action set $\mathcal{A}$ for an agent. By performing an action $a \in \mathcal{A}$, the agent can move from state s_k to state s_{k+1}, and the agent gets a reward $R(s_{k+1})$. The goal of the agent is to maximize its total reward. It does this by learning which action is optimal for each state. For this, value function $Q(s, a)$ which maps from state-action to total reward and can be approximated using a function approximator (e.g., multi-layered perception, memory based system, radial basis functions, look-up table, etc.) Before learning has started, $Q(s, a)$ are initialized to a fixed value. Then, the agent gets the reward values which are calculated for each combination of a state s from $\mathcal{S}$, and action a from $\mathcal{A}$. The core of the algorithm is a value iteration update. It assumes the old value and makes a correction based on the new information.

$$Q(s_k, a_k) \leftarrow \underbrace{Q(s_k, a_k)}_{\text{old value}} + \underbrace{\alpha_k}_{\text{learning rate}} \left[\overbrace{\underbrace{R(s_{k+1})}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_{a_{k+1}} Q(s_{k+1}, a_{k+1})}_{\text{max future value}}}^{\text{learned value}} - \underbrace{Q(s_k, a_k)}_{\text{old value}} \right] \tag{4.18}$$

where α_k ($0 < \alpha_k \leq 1$) is the learning rate, the discount factor γ is such that $0 \leq \gamma < 1$. The optimal policy therefore is the mapping from states to actions that maximizes the sum of the reinforcements.

For learning optimal policy to stabilize walking, we constructed a simulation environment that has a source to disturb the robot, i.e. the uneven ground (see Figure 4.20) which contains some small objects with size less than 1 cm. The state $\mathcal{S}$ are the state of the robot, i.e. the angle of the chest (from the inertial sensor data), the gyro data, and the FSR data in the feet. We implemented the algorithm using the look-up table. The angles of the chest are sampled in thirty states (2° in [-30°, 30°]); the gyro data is sampled in three states (negative, zero, and positive); the FSR has four states (touched, untouched for two feet), so the number of total states is 360. The actions are added offseting the planned CoM, and sampled to 5 different values. Therefore, there are 1800 values in the Q matrix, and the look-up table is efficient enough. Note that we treated the controller in the forward-backward and left-right direction separately. The reward is defined as follow:

$$R(s) = \begin{cases} -1 & |\theta| > \Theta; \\ 0 & \text{otherwise.} \end{cases} \tag{4.19}$$

where θ is the angle of the chest, Θ is the maximum angle where the robot doesn't fall down, i.e. 30° in our case. It means that the robot gets a negative reward when it falls down.

The Q-learning has the following steps:

1. Initialize the Q to 0. In other words: Set for all (s, a) where $s \in \mathcal{S}$ and $a \in \mathcal{A}$,

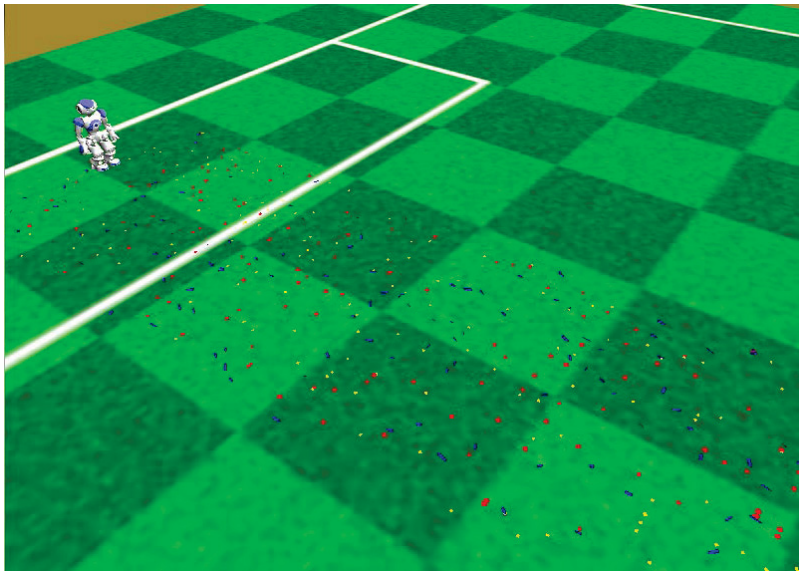


Figure 4.20: Learning to balance during walking, the robot walks around the field, and learns to balance using the distribution caused by small objects (size less than 1 cm) on the ground.

$Q(s,a)$ to 0. Set k, the number of iterations, to 0. We will run the algorithm for k_{max} iterations, where k_{max} is chosen to be a sufficiently large number. Set A and B, the step size constants, set to a positive number less than 1. Set exploratory constants ϵ to a positive number. The robot starts to walk.

2. Let the current state be s_k. Let $a_k^* \leftarrow \arg\max_{a \in \mathcal{A}} Q(s_k, a)$. Select action a_k^* as the action a_k with a probability $1 - \epsilon/k$; or select a_k from $\mathcal{A}$ uniformly at random.

3. Execute action a_k. Let the next state be s_{k+1}. Calculate the immediate reward $R(s_{k+1})$ in equation (4.19). If the robot falls down, reset the robot and restart the walk.

4. calculate $\alpha_k = A/(B+k)$, then update $Q(s_k, a_k)$ using equation (4.18).

5. Increment k by 1. If $k < k_{max}$, go to Step 2. Otherwise, go to Step 6.

6. For each $s \in S$, select
$$p(s) = \arg\max_{a \in \mathcal{A}} Q(s,a) \tag{4.20}$$
The policy (solution) generated by the algorithm is p, stop.

At the beginning of the learning phase, the robot reacts to the sensor data randomly, it falls down very often. After thousands of trials, the robot is able to take advantage of the sensor data to keep its balance.

With the learned policy, the real robot is able to walk stably on the field. Compared with the stabilizers we developed in section 4.2.1, the learned controller integrates different sensors. Furthermore it doesn't depend on the model of the robot. In the next step, we may also include different actions into the RL, e.g. adjusting the step size.

In this section, we focused on developing the biped walk for NAO. A closed loop walk engine based on the ZMP was implemented. Furthermore, we used the improved SimSpark to improve the performance of the walk. Techniques including the Evolution Algorithm and the Reinforcement Learning are used. Finally, our NAOs are able to walk stably on even ground, with speed of 20 cm/s in forward/backward directions, 10 cm/s sideways and turn 60 °/s at the same time.

4.3 Summary

In this chapter, we investigated how to develop motions in the simulation for the humanoid robot. The simulator developed in chapter 3 was used as a research platform.

First, we optimized the speed of the keyframe motion, i.e. standing up, in the simulation; and tested it in the real robot. We repeat the simulator optimization and keyframe motion optimization until the real robot stands up successfully with parameters optimized in the simulation. Then, the whole keyframe motion was evolved in the simulation and transferred to the real robot. Because of lack of feedback from sensors, the keyframe motion in real robot is not as stable as in simulation. Further research is still needed for using in real applications.

For dynamic motion, i.e. walking, we use the ZMP model as stability criterion and the LIP model to approximate the dynamics of the robot. The simulation enables us to use evolution to optimize parameters for walking speed. Although we got slower speed in the improved simulator than in the original SimSpark, the resulting parameters in the original SimSpark can not be applied to real robot due to the high frequency steps. The improved simulator allows us to optimize the foot trajectory for saving energy, and the resulting foot trajectory consumes less energy than our predefined one. Furthermore, we take advantage of the simulator by using the Reinforcement Learning instead of the PID controllers to maintain balance during walking, and the learned policy is transferred to the real robot.

The experimental results show that we can develop and improve motions in the simulation and transfer them to the real robot later without adjustment. The resulting walk is not the fastest one, but it is faster than the walk from manufacturer, and more important is that it is energy effective. However, the motions do not work well in new environment, e.g. another playing field. In future research, we plan to investigate online machine learning to deal with this problem.

5 Conclusion and Future Work

> The only way of finding the limits of the possible is by going beyond them into the impossible.
>
> — A.C. Clarke

5.1 Conclusion

Physical simulation is an effective and practical method to study and explore real world problems. The ideal robot development scenario is developing algorithms in the simulation first, and then moving them to real robots without having to make any change. However, there were no examples of generating complex behaviors for the humanoid robot in a simulation and transferring them to reality successfully. Simulations can give valuable results for robotics only in close connection to real robots.

In this paper, we investigated how to create a mechanism that provides a smooth gradient to migrate the robot from a purely virtual world to an entirely real implementation. This raises questions like, *How to measure the outcome of the simulation? How to improve the accuracy of the current simulator for the humanoid robot?* and *How to develop a control program in the simulation and transfer it to the real robot?*

We have developed a framework for running robots both in real and simulated settings. In chapter 1, we briefly described the architecture and modules of our RoboCup team – Nao Team Humboldt – which participates in both the Standard Platform League and the Simulation 3D Leagues with the same source code. Thanks to the flexibility of our framework, we can develop and test our code in the simulation, that is the SimSpark, and apply it to the real robot later. However, the reality gap between simulation and reality is the main problem for us to be able to benefit directly from the simulation.

In chapter 2, we discussed the methods to evaluate the simulation and evaluated the SimSpark in the model level and the results level. In the model level we found two main problems 1) some models of the robot contain approximations or errors, e.g. mass, motor; 2) the parameters of the simulation engine affects the results. For evaluating results directly, we use both the internal sensors of the robot and the external sensors. The Kinect was used as the motion capture system. Hence, we can measure the outcome of the simulation, and use it to improve the simulation later.

Chapter 3 describes our work for improving the realism of the SimSpark. After investigating the NAO documentation, we refined the robot model by using accurate information, modeled and implemented the NAO sensors, including the accelerometer, RGB camera and the lines in virtual vision. For the actuator, we developed a servo

model with backlash and power consumption, which behave similarly to servo in the real NAO. The parameters of the simulator are optimized by the Evolutionary Algorithm. We compared data from the motion capture in reality with the simulation results as fitness. The improved SimSpark is evaluated with keyframe motions which failed in the original SimSpark.

In chapter 4, we investigated how to develop motions in the improved SimSpark for the NAO robot. First, the speed of the keyframe motion, i.e. standing up, is optimized in the simulation, and tested in the real robot; then a stand up motion based on the keyframe was generated in the simulation and transferred to the real robot. For dynamic motion, i.e. biped walking, we use the ZMP model as stability criterion and the Linear Inverted Pendulum model to approximate the dynamics of the robot, and evolution to optimize the parameters. Furthermore we use Machine Learning to learn foot trajectory and maintaining balance. After testing and optimizing in the simulation, the real NAO can walk stably at a speed of 20 cm/s.

In conclusion, the main contributions of this thesis are:

- A seamless migration of code between real and simulated robots.
- Evaluation of the simulation results by data from the robot sensor and the external sensor.
- Modeling sensors and actuators for the humanoid robot simulation, and optimizing the parameters with result evaluations.
- Developing and optimizing motions for the humanoid robot in the simulation, and transferring them to the real robot.

All these improvements together allow robot software development in a more efficient way. We can develop motions in the simulation and transfer them to the real humanoid robot later without making any adjustments. Although, the resulting motions still can be optimized (see Figure 2.30), this demonstrates the simulation can be used as an evolution and learning platform for improving skills of real robots. Some features developed in this work have been used in the official SimSpark, including the accelerometer, RGB camera and the lines in virtual vision. Some features could be included in the future version of SimSpark, such as the servo model with backlash and power consumption.

In the past RoboCup competitions, as an important research tested field for this paper, we achieved some good results in both leagues. In the Standard Platform League, we won second place in the RomeCup 2010, first place in the SETN 2010, fourth place in the German Open 2010, and first place in the Iran Open 2011. Based on the work of this thesis in particular, we won the Open Challenge in the RoboCup 2012. In the Simulation 3D League, we won first place in the German Open 2010, second place in the RoboCup 2010, first place in the AutCup 2010, first place in the RC4EW 2010, fourth place in the German Open 2011, and fourth place in the Iran Open 2011.

5.2 Discussion

During the research for this thesis, we participated in several RoboCup competitions in the Simulation League and the Standard Platform League. We participated in both leagues at the same time in the German Open 2010, the German Open 2011 and the RoboCup 2011 with the same source code. In this section, we will discuss the effects of this thesis in the competition. The performance of our team in the Simulation League and the Standard Platform League are compared.

The German Open 2010 was the first time we played on two fields (virtual and real) at the same time. A lot of modules, e.g. walking, self localization, were reused in the simulation with some parameter adjustments. We optimized the parameters of walking by using the Evolution Algorithm. It was one of the fastest walk (65 cm/s) in the simulation at that time. However, we had problems making the real robot walk stably at the beginning. Our robot could walk after a lot of the parameters were adjusted for the particular field. However the speed was approximately 10 cm/s and much slower than those of the top teams. We also had trouble with the image processing in the venue, therefore self localization in the simulation worked better than in the real robot, since the percepts from its vision were not as good as the simulated ones. This was the best competition we had when we participated in two leagues at the same time.

In the German Open 2011 and the RoboCup 2011, we had a lot of problems with the real robot. We didn't managed to solve the low level (hardware related) issues. In the Simulation League, we were also not the strongest. Some code changes for the real robot affected the simulated robot, i.e. preview in walking. The preview control was introduced to smooth the trajectory of the center of mass, but it also introduced delay between decision and action. We could use robot's previewed position to minimize the effects of the delay, but the robot reacts slower to the moving ball anyway. Because the ball is especially difficult to predict, it was touched by robots all the time. One second delays didn't affect the robot a lot in slow games, but it did affect the robot a lot in fast games. Other simulation teams became stronger and performed better.

Another problem of us is adjustment before the game, including calibrations. The time is very limited in the competition, but we had to adjust teams for different leagues just before the game. If development and testing were done before the competition, and calibration could be easily done, it would be really fun to have one team play in both the virtual and the real fields. Although the game from different leagues looks very different, we hope it will become more similar in the future, and can improve or find bugs in the simulation that affects the real robot positively.

The code of whole team includes seven main modules (see Figure 1.6). The details of implementation are described in our team descriptions [9–11]. Some different modules were used for different leagues. The most interesting part is the modules which were used in both leagues. Next, we discuss these modules in three categories: cognition (perception and modeling), behavior, and motion.

Perception We use exactly the same modules in perception, except for the computer vision. In order to use the same modules for modeling, the virtual vision and the computer vision processing module provide the same percepts, including ball percept, goal percept, line percept and other robot percept. Each percept contains position information of objects in the camera coordinate system. However recognizing other robots is difficult in the real robot, due to the limited computation resource for image processing.

The most important modules in modeling are the Self Locator and the Ball Locator, as they provide important information for the game: *"where am I"* and *"where is the ball"*. We investigated different methods using various perspectives in the previous years [9–11], including classical Bayesian approaches with Kalman filters and the Monte-Carlo methods, and the Constraint based techniques. Kalman filters make certain assumptions about the environment (linearity of the model, Gaussian noise), Monte-Carlo methods with particle filters are restricted by high computational demand and thus, low dimensionality. The Constraint techniques are computationally cheap using interval arithmetic, and can be easily communicated allowing for cooperative localization, but they have to handle inconsistent data [30].

Behavior We use XABSL [38] to develop the high level of behavior for our robot soccer player. XABSL describes the behaviors of an agent with a set of finite state machines that are organized in a hierarchy. Each state machine is called an *option* and the set of options is called *option graph*. We use the same *option graphs* in the middle and low levels, such as skill *options*, e.g. walk to ball, kick the ball to goal, etc. These skill *options* work quite well in both the simulation and the real robot. We investigated how to formalize soccer theory so that specification and execution is possible in our robot. The advantage is clear: theory abstracts from hardware and from specific situations in leagues. However, to optimize the performance in different leagues and follow different rules in different leagues, we developed the top *option* differently.

Due to different number of players between the Simulation League and the Standard Platform League, we implemented different role decision *option graphs*. We can test behaviors for the Standard Platform League in the simulation. And we could also use behaviors for the Simulation League in the real robot, when we have same field and same number of players as the simulation. Next, we may abstract this knowledge to a higher level: independently of the field size and the number of players. However, it is difficult, since the game styles are totally different on different sized fields. For example, the robots can shoot directly on a smaller field, but have to dribble and pass the ball on a bigger field.

At the moment, coordination of our robots relies on communication between the robots. Each robot shares its high level decision making (e.g. go to the ball or not) and some important modeling data (e.g. the ball position on the field). Although the communication channels are different in the simulation and the real robot, the same protocol is used, e.g. the information regarding the ball and its global position on the field are communicated. We have different roles assigned to each robot. Usually, we have one active robot as the attacker or defender, while the others play as supporters. The active robot

is the one who can get the ball in the shortest time; the others are passive. Collaboration between the passive robots can be achieved by making use of formations, which are a principal concept for a soccer team since they distribute the players on the field well. In our implementation, the positioning mechanism is based on a method called Situation Based Strategic Positioning (SBSP)[63].

For the active robot who is processing the ball, it has to decide how to deal with ball, e.g. kicking or dribbling, and where to kick and dribble it to. Because situations in the soccer game are very dynamic and complex, there are infinite possible sub-options for XABSL. We created a module in C++ code, which provides an optimized attack direction in the current situation. At the moment, the potential field is implemented here. The opponent goal is treated as an attractive source. The team's own goal and opponent robots are treated as a rejection source. The results of the potential field give the best direction to go in the current situation.

In short, we formalized soccer theory in the XABSL options and use them in both leagues, but some specialized options are implemented for different rules. This shows the power of XABSL, and the benefits of a common platform.

Motion In both the Simulation League and Standard Platform League we use the unique motion module. However, the motion module uses different parameters and configurations in different platforms. At the moment, only the head motion performs exactly the same in both platforms. Because of the limited view range of the camera, the robot should look around smartly. For the game, robot should be able find the ball fast and track it. Our motion provides an abstract interface for these tasks. The behavior requests which point look at in 3D, and the head motion is executed to minimize the difference between the current looking direction and the requested one. A higher level searching and planning algorithm is implemented in the behavior part, which can use results of image processing. For fast and efficient searching, the idea of attention is implied here.

As we stated in chapter 2, the robot can not use the same keyframes in the official SimSpark as it does in the real robot. It is necessary to create different keyframes to do this, fortunately it is only necessary in stand up motion. In chapter 4 we talked a lot about walking, in the game we just use different parameters. However, the preview control walk makes the robot react slowly. In other words, we have to improve the reality of SimSpark for the competition, or we have to improve the walking in this virtual world to win. We have developed a dynamic kick, which can adjust the kicking foot to any given kicking direction dynamically [51, 75]. Therefore our robot does not need to adjust his position precisely before kicking. However, it doesn't work in the Simulation League at all, because the physical model of the ball in the simulation is very different from the real one. The robot has to hit a particular point on the ball to kick it far away. This is one of the necessary improvement of the SimSpark. Consequentially, a lot of teams only dribble the ball during a game.

The Future of the Simulation League In the Standard Platform League, it is not easy for the team to switch from the NAO to another platform. This is because it be more

costly. In the Simulation League, it is relatively easy to introduce new robots. All that is needed is to create the robot models and the teams need to adjust the code for the new models. In the future, different robots will play together in the same game, this is will introduce heterogeneous robot research in the 3D Simulation League just as it did in the 2D Simulation League. It will be pre-research for cooperation and competition between different robots in the hardware leagues.

However, it is not easy to create rules that are fair for the different robot models. The discussion is still open and there are some proposals: the exact configurations of the robots is unknown but the range is known (like 2D simulation); the configurations of the robots are released just a few days before competition, that means teams will not have too much time to optimize for specialized robot models; the team can create their own robot models, but under the general rules, this is like the Humanoid League, e.g. the height of the robot, the mass of the robot and the height of center of mass configuration should be in a given range.

Because of the difficulty of improving the realism of simulation, some teams proposed that we should not be limited by the real robots. This way, we can do advanced research on things such as team coordination, opponent modeling. Both topics are not considered in real robots today.

We do not completely agree with this. First of all, as stated in the beginning, the RoboCup is divided into different leagues for solving different sub-tasks, but the teams in the different leagues are solving the same problems, such as self localization, motion generation, etc. Secondly, if some teams are only interested in advanced (high level) research, we already have the 2D simulator, which has been developed for several years and proved to be a good research platform for strategies (high level). Furthermore, when we take the vision of the 2050, the robot has to play against a human champion. The research done in the Simulation League has to be transferred to the real robot. It will be easier if we start earlier. Lastly but not least, the reality of the SimSpark is improved, more teams from other leagues will like to use it and contribute to the SimSpark. This will contribute to the whole RoboCup community.

5.3 Work for Further Improvements

To continue to improve the arena we can take the following approaches in the following areas:

Realistic Physics Engine The ODE is the most widely used rigid-body dynamic implementation for the robot simulation, but it has some specs (e.g. collision, friction) which limit the physical attributes. This is unsatisfactory for the robot applications. Extending the ODE [21] or using other physics engines, e.g. bullet [1], may solve these basic limitations.

Realistic Image Sensor The image generated by the computer is different from the data captured by the camera. It makes transferring computer the vision algorithm between the simulation and the reality difficult. Furthermore, it affects the higher level of cognition, such as world modeling, and decision making. The SimRobot is already able to simulate common image disturbances, i.e. the rolling shutter effect and motion blur [59].

Statistical Modeling The sensor inputs and the actuator output in real robots are noisy and collect biased data. If we can get accurate statistics and integrate the results into the simulator, the transfer of control program between the simulation and the real robot may be even easier, because there are unrealistic assumptions about the probability density function of the sensor noise in the current system. However, to do this we may need accurate sensors to capture the data in reality. Maintenance of nonparametric estimators over data streams [6] can be another approach.

Online Machine Learning The perfect model is never expected to be found, since the model is an abstraction of the real system. The optimal policy learned in the simulation cannot be guaranteed as optimal in reality. When a robot can learn online, it can improve the policy in reality after learning them in the simulation. Hemker [36] included a real humanoid robot in the optimization as hardware-in-the-loop to optimize low dimensional parametrized walking.

Appendix

1 NAO robot specifications

1.1 Specification of the servo motors

There are two types of motors and four types of speed reduction ratio in NAO V3 and for each motor.

Table A.1: Specifications of two types motor in NAO from manufacture.

Model		RE-Max 24	RE-Max 17
No load speed	rpm	8000	11900
Stall torque	mNm	59.5	15.1
Nominal speed	rpm	6330	8810
Nominal torque	mNm	12.3	3.84
Terminal resistance	Ω	6.44	23.1
Torque constant	mNm/A	21.3	16.6
Speed constant	rmp/V	448	577

Table A.2: Specifications of four types joint in NAO.

Type		1	2	3	4
Motor		RE-Max 24	RE-Max 17	RE-Max 24	RE-Max 17
Speed reduction ratio		201.3	150.27	130.85	173.22
No load speed	rpm	39.7	79.2	61.1	68.7
Stall torque	Nm	12.0	2.3	7.8	2.6
Nominal speed	rpm	31.4	58.6	48.4	50.9
Nominal torque	Nm	2.5	0.6	1.6	0.7
Terminal resistance	Ω	6.44	23.1	6.44	23.1
Torque constant	Nm/A	4.3	2.5	2.8	2.9
Speed constant	rmp/V	2.2	3.8	3.4	3.3
Joints		HipYawPitch HipRoll AnkleRoll	HeadYaw ShoulderPitch ElbowYaw	HipPitch KneePitch AnklePitch	HeadPitch ShoulderRoll ElbowRoll

Bibliography

[1] Bullet physics library. URL `http://bulletphysics.org`. Last visited at 07.02.2013.

[2] Aldebaran Robotics. *NAO Software 1.12 Documentation*. URL `http://www.aldebaran-robotics.com`, Last visited at 07.02.2013.

[3] Mihai Anitescu and Gary D Hart. A Constraint-Stabilized Time-Stepping Approach for Rigid Multibody Dynamics with Joints , Contact and Friction. *International Journal for Numerical Methods in Engineering*, pages 1–6, 2002.

[4] Australian Defence Simulation Office. *Simulation Verification , Validation and Accreditation Guide*. Commonwealth of Australia, 2005.

[5] Sven Behnke. Online trajectory generation for omnidirectional biped walking. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006*, pages 1597–1603, May 2006. ISBN 0-7803-9505-0.

[6] Björn Blohsfeld, Christoph Heinz, and Bernhard Seeger. Maintaining nonparametric estimators over data streams. In *BTW*, pages 81–85. Citeseer, 2004.

[7] Adrian Boeing. Physics abstraction layer. URL `http://www.adrianboeing.com/pal/index.html`. Last visited at 07.02.2013.

[8] Adrian Boeing and Thomas Bräunl. Evaluation of real-time physics simulation systems. *Computer Engineering*, 1(212), 2007.

[9] Alexander Borisov, Alireza Ferdowsizadeh, Christian Mohr, H. Mellmann, Martin Martius, Thomas Krause, Tobias Hermann, Oliver Welter, and Yuan Xu. NAO-Team Humboldt 2009. In *RoboCup 2009: Robot Soccer World Cup XIII*, 2009.

[10] Hans-Dieter Burkhard, Verena Hafner, Florian Holzhauer, Thomas Krause, Heinrich Mellmann, Claas-Norman Ritter, Oliver Welter, and Yuan Xu. NAO-Team Humboldt 2010. In *RoboCup 2010: Robot Soccer World Cup XIV*, 2010.

[11] Hans-Dieter Burkhard, Thomas Krause, Heinrich Mellmann, Claas-Norman Ritter, Yuan Xu, Marcus Scheunemann, Martin Schneider, and Florian Holzhauer. NaoTH 2011: The RoboCup Team of Humboldt-Universitat zu Berlin. In *RoboCup 2011: Robot Soccer World Cup XV*, 2011.

[12] P Cignoni. Metro: measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, 1998.

[13] Steven H. Collins, Martijn Wisse, and Andy Ruina. A three-dimensional passive-dynamic walking robot with two legs and knees. *The International Journal of Robotics Research*, 20(7):607–615, July 2001. ISSN 0278-3649.

[14] Pascal Cominoli. Development of a physical simulation of a real humanoid robot. Master's thesis, BIRG, Logic Systems Laboratory (LSL), School of Computer and Communication Sciences, Swiss Federal Institute of Technology, Lausanne, February 2005.

[15] NVIDIA Corporation. Physx. URL `http://www.geforce.com/hardware/technology/physx`. Last visited at 07.02.2013.

[16] Cyberbotics Ltd. Webots: the mobile robotics simulation software. URL `http://www.cyberbotics.com`. Last visited at 07.02.2013.

[17] S. Czarnetzki, G. Jochmann, and S. Kerner. Nao Devils Dortmund. In *RoboCup 2011: Robot Soccer World Cup XV*. Springer, 2011.

[18] Julian Andrew de Marchi. *Modeling of dynamic friction, impact backlash and elastic complicance nolinearities in machine tools, with applications to asymmetric viscous and kinetic friction identification*. PhD thesis, Rensselær Polytechnic Institute, Troy, New York, December 1998.

[19] Rosen Diankov. *Automated Construction of Robotic Manipulation Programs*. PhD thesis, Carnegie Mellon University, Robotics Institute, August 2010.

[20] Evan Drumwright. A fast and stable penalty method for rigid body simulation. *IEEE transactions on visualization and computer graphics*, 14(1):231–40. ISSN 1077-2626.

[21] Evan Drumwright, John Hsu, Nathan P. Koenig, and Dylan A. Shell. Extending open dynamics engine for robotics simulation. In Noriaki Ando, Stephen Balakirsky, Thomas Hemker, Monica Reggiani, and Oskar von Stryk, editors, *SIMPAR*, volume 6472 of *Lecture Notes in Computer Science*, pages 38–50. Springer, 2010. ISBN 978-3-642-17318-9.

[22] Uwe Dueffert and Jan Hoffmann. Reliable and precise gait modeling for a quadruped robot. *RoboCup 2005: Robot Soccer World Cup IX*, 2006.

[23] Frank Dylla, Alexander Ferrein, Gerhard Lakemeyer, Jan Murray, Oliver Obst, Thomas Röfer, Frieder Stolzenburg, Ubbo Visser, and Thomas Wagner. Towards a league-independent qualitative soccer theory for robocup. In D. Nardi, editor, *RoboCup 2004, LNAI3276*, pages 29–40, Berlin Heidelberg New York, 2005. Springer.

[24] R Featherstone. *Rigid Body Dynamics Algorithms*. Springer, 2008. ISBN 978-0-387-74314-1.

[25] Jaime J. Fernandey. The genentic programming notebook. URL `http://www.geneticprogramming.com/`. Last visited at 07.02.2013.

[26] Marc Freese. Virtual robot experimentation platform. URL `http://www.v-rep.eu/`. Last visited at 07.02.2013.

[27] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc., 2008. ISBN 9780596516130.

[28] Bill Gates. A robot in every home. *Scientific American*, 1:1–8, January 2007.

[29] B. Gerkey, R.T. Vaughan, and A.. Howard. The player/stage project: Tools for multirobot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics*, pages 317 – 323, Coimbra, Portugal, 2003.

[30] D. Göhring. *Constraint based world modeling for multi agent systems in dynamic environments*. PhD thesis, Humboldt Universtiy Berlin, 2009.

[31] D. Gouaillier, V. Hugel, P. Blazevic, C. Kilner, J. Monceaux, P. Lafourcade, B. Marnier, J. Serre, and B. Maisonnier. The nao humanoid: a combination of performance and affordability. *ArXiv e-prints*, July 2008.

[32] C. Graf and T. Röfer. A center of mass observing 3D-LIPM gait for the RoboCup Standard Platform League humanoid. In *Robot Soccer World Cup XV. Lecture Notes in Artificial Intelligence*, Istanbul, 2011. Springer.

[33] Rodrigo Guerra, Joschka Boedecker, N. Michael Mayer, Shinzo Yanagimachi, Yasuji Hirosawa, Kazuhiko Yoshikawaka, Masaaki Namekawa, and Minoru Asada. Introducing physical visualization sub-league. In *RoboCup 2007: Robot Soccer World Cup XI*, Atlanta, USA, July 2007.

[34] Daniel Hein. Simloid – evolution of biped walking using physical simulation. Master's thesis, Humboldt-University Berlin, Germany, January 2007.

[35] Daniel Hein, Manfred Hild, and Ralf Berger. Evolution of biped walking using neural oscillators and physical simulation. In *RoboCup 2007: Robot Soccer World Cup XI*, Atlanta, USA, July 2007.

[36] T. Hemker, M. Stelzer, O. von Stryk, and H. Sakamoto. Efficient Walking Speed Optimization of a Humanoid Robot. *The International Journal of Robotics Research*, 28 (2):303–314, February 2009. ISSN 0278-3649.

[37] KA De Jong. *Evolutionary computation: a unified approach*. The MIT Press, 2006. ISBN 0262041944.

[38] M. Lötzsch Jüngel, M. Risler, and M. XABSL - A Pragmatic Approach to Behavior Engineering. In *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, pages 5124–5129, Beijing, China, 2006.

[39] Shuuji Kajita, Fumio Kanehiro, Kenji Kaneko, Kiyoshi Fujiwara, Kazuhito Yokoi, and Hirohisa Hirukawa. Biped walking pattern generation by a simple three-dimensional inverted pendulum model. *Advanced Robotics*, 17(2):131–147, 2003.

[40] Shuuji Kajita, Hirohisa Hirukawa, Kazuhito Yokoi, and Kensuke Harada. *Humanoid Robots*. Ohm-sha, Ltd, 2005.

[41] Fumio Kanehiro, Kiyoshi Fujiwara, Shuuji Kajita, Kazuhito Yokoi, Kenji Kaneko, Hirohisa Hirukawa, Yoshihiko Nakamura, and Katsu Yamane. Open architecture humanoid robotics platform. In *Proceeding of the 2002 IEEE International Conference on Robotics and Automation*, pages 24 – 30, Washington, DC, May 2002.

[42] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The robot world cup initiative. In W. Lewis Johnson and Barbara Hayes-Roth, editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 340–347, New York, 5–8, 1997. ACM Press. ISBN 0-89791-877-0.

[43] J Kleijnen. Verification and validation of simulation models. *European Journal of Operational Research*, 82(1):145–162, April 1995. ISSN 03772217.

[44] N. Koenig and A. Howard. Gazebo - 3d multiple robot simulator with dynamics, 2004. URL `http://playerstage.sourceforge.net/gazebo/gazebo.html`. Last visited at 07.02.2013.

[45] Marco Kögler. Simulation and visualization of agents in 3d environments. Master's thesis, Koblenz-Landau University, Germany, 2003.

[46] Marco Kögler and Oliver Obst. Simulation league: The next generation. In Daniel Polani, Andrea Bonarini, Brett Browning, and Kazuo Yoshida, editors, *RoboCup 2003: Robot Soccer World Cup VII*, Lecture Notes in Artificial Intelligence, Berlin, Heidelberg, New York, 2004. Springer.

[47] Tim Laue and Matthias Hebbel. Automatic parameter optimization for a dynamic robot simulation. *RoboCup 2008: Robot Soccer World Cup XII*, pages 121–132, 2009.

[48] Tim Laue, Kai Spiess, and Thomas Rofer. Simrobot | a general physical robot simulator and its application in robocup. In *RoboCup 2005: Robot Soccer World Cup IX*, Osaka, Japan, June 2005.

[49] Beatriz León, Stefan Ulbrich, Rosen Diankov, Gustavo Puche, Markus Przybylski, Antonio Morales, Tamim Asfour, Sami Moisio, Jeannette Bohg, James Kuffner, and Rüdiger Dillmann. Opengrasp: A toolkit for robot grasping simulation. In Noriaki Ando, Stephen Balakirsky, Thomas Hemker, Monica Reggiani, and Oskar von Stryk, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 6472 of *Lecture Notes in Computer Science*, pages 109–120. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-17318-9.

[50] Norbert Michael Mayer, Joschka Boedecker, Rodrigo da Silva Guerra1, Oliver Obst, and Minoru Asada. 3d2real: Simulation league finals in real robots. In *RoboCup 2006: Robot Soccer World Cup X*, Bremen, Germany, June 2006.

[51] Heinrich Mellmann and Yuan Xu. Adaptive motion control with visual feedback for a humanoid robot. In *the Proceedings of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Taipei, 2010.

[52] Heinrich Mellmann, Yuan Xu, Thomas Krause, and Florian Holzhauer. Naoth software architecture for an autonomous agent. In *International Workshop on Standards and Common Platforms for Robotics (SCPR 2010)*, Darmstadt, November 2010.

[53] Microsoft Corporation. Microsoft robotics studio. URL `http://msdn.microsoft.com/robotics`. Last visited at 07.02.2013.

[54] BV Mirtich. *Impulse-based dynamic simulation of rigid body systems*. PhD thesis, University of California at Berkeley, 1996.

[55] S. Mojon. Realization of a physic simulation for a biped robot. Semester Project at BIRG laboratory, Swiss Federal Institute of Technology, 2003.

[56] DJ Murray-Smith. The inverse simulation approach: a focused review of methods and applications. *Mathematics and computers in simulation*, 53:239–247, 2000.

[57] Masaki Ogino. *Embodiment Approaches to Humanoid Behavior*. PhD thesis, Department Of Adaptive Machine Systems, Osaka University, Japan, January 2005.

[58] DK Pace. Modeling and simulation verification and validation challenges. *Johns Hopkins APL Technical Digest*, pages 163–172, 2004.

[59] Dennis Pachur, Tim Laue, and Thomas Röfer. Real-time Simulation of Motion-based Camera Disturbances. In *RoboCup 2008: Robot Soccer World Cup XII*, 2009.

[60] Georg Pelz. *Mechatronic systems: modelling and simulation with HDLs*. Wiley, 2003.

[61] C Pepper, S Balakirsky, and C Scrapper. Robot simulation physics validation. In *Proceedings of the 2007 Workshop on Performance Metrics for Intelligent Systems*, pages 97–104. ACM, 2007.

[62] K.R. Popper. *The logic of scientific discovery*. Psychology Press, September 2002. ISBN 0415278430.

[63] L Reis and Nuno Lau. Fc portugal team description: Robocup 2000 simulation league champion. In G. Stone, P. and Balch, T. and Kraetzschmar, editor, *RoboCup 2000: Robot Soccer World Cup IV*. Springer, 2001.

[64] T. Röfer, J. Brose, D. Göhring, M. Jüngel, T. Laue, and M. Risler. GermanTeam 2007 - The German national RoboCup team. In *RoboCup 2007: Robot Soccer World Cup XI Preproceedings*. RoboCup Federation, 2007.

[65] Markus Rollmann. Spark – a generic simulator. Master's thesis, Koblenz-Landau University, Germany, 2004.

[66] Randi J Rost. *OpenGL® Shading Language, Second Edition*. Addison Wesley Professional, 2006. ISBN 9780321334893.

[67] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[68] Robert Shannon and James D. Johannes. Systems simulation: The art and science. *Systems, Man and Cybernetics, IEEE Transactions on*, 6(10):723 –724, oct. 1976. ISSN 0018-9472.

[69] Russell Smith. *Open Dynamic Engine User Guide*, 2006. URL `http://www.ode.org`, Last visited at 07.02.2013.

[70] Sony. Sony launches four-legged entertainment robot, May 1999. URL `http://www.sony.net/SonyInfo/News/Press_Archive/199905/99-046/`. Last visited at 07.02.2013.

[71] The RoboCup Federation. The official robocup website. URL `http://www.robocup.org`. Last visited at 07.02.2013.

[72] Miomir Vukobratovic and Branislav Borovac. Zero-moment point- thirty five years of its life. *International Journal of Humanoid Robotics*, 1(1):157–173, 2004.

[73] Jijun Wang and Stephen Balakirsky. *USARSim – A Game-based Simulation of the NIST Reference Arenas*. University of Pittsburg, May 2005. URL `http://sourceforge.net/projects/usarsim/`, Last visited at 07.02.2013.

[74] Yuan Xu and Hans-Dieter Burkhard. Narrowing reality gap and validation: Improving the simulator for humanoid soccer robot. In *Concurrency, Specification and Programming CS&P'2010*, Helenenau, Germany, September 2010.

[75] Yuan Xu and Heinrich Mellmann. Adaptive motion control: Dynamic kick for a humanoid robot. In *the Proceedings of the 33rd Annual German Conference on Artificial Intelligence*, Karlsruhe, Germany, 2010.

[76] Yuan Xu, Heinrich Mellmann, and Hans-Dieter Burkhard. An approach to close the gap between simulation and real robots. In Ando, Balakirska, Reggiani, and von Stryk, editors, *2nd International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR)*, Darmstadt, November 2010.

[77] Feng Xue, Xiaoping Chen, Jinsu Liu, and Daniele Nardi. Real Time Biped Walking Gait Pattern Generator for a Real Robot. In *Robot Soccer World Cup XV. Lecture Notes in Artificial Intelligence*. Springer, 2011.

[78] J.C. Zagal, J.R. del Solar, and P. Vallejos. Back to reality: Crossing the reality gap in evolutionary robotics. In *Proceedings of the 5th IFAC/EURON Symposium on Intelligent Autonomous Vehicles*, 2004.

[79] Juan Cristóbal Zagal and Javier Ruiz del Solar. Uchilsim: A dynamically and visually realistic simulator for the robocup four legged league. In *RoboCup Symposium 2004*, 2004.

[80] Juan Cristóbal Zagal and Javier Ruiz-del Solar. Learning to kick the ball using back to reality. In Daniele Nardi, Martin Riedmiller, Claude Sammut, and José Santos-Victor, editors, *RoboCup 2004: Robot Soccer World Cup VIII*, volume 3276 of *Lecture Notes in Computer Science*, pages 335–346. Springer Berlin / Heidelberg, 2005.

List of Figures

List of Tables

Selbständigkeitserklärung

Ich erkläre hiermit, dass

- ich die vorliegende Dissertationsschrift „From Simulation to Reality:
  Migration of Humanoid Robot Control" selbständig und ohne unerlaubte Hilfe angefertigt habe;
- ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe oder einen solchen besitze;
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin vom 17. Januar 2005 (zuletzt geändert am 13. Februar 2006 - Nr. 34/2006) bekannt ist.

Berlin, den 11. Februar 2013 Yuan Xu

Acknowledgement

I am heartily thankful to my supervisor, Prof. Dr. Hans-Dieter Burkhard, whose encouragement, supervision and support enabled me to research on this interesting topic.

I want to thank all team members in Nao Team Humboldt, for the wanderful work on NAO and RoboCup events around the world. I also would like to make a special reference to Mr. Heinrich Mellmann who is the team leader. Without corporation and discussions with him, I could not have gotten the work done.

I have had the pleasure of working with Prof. Dr. Yingzi Tan in Southeast University China, who lead me into RoboCup and helped me to drive things forward. I also want to thank Prof. Dr. Verena Hafner for the hints to make this work clearer to understand.

I am grateful to China Scholarship Council for the finalcial support, I am also thankful to Chinese Educational Service in Germany for their assistance.

Where would I be without my family. I like to thank my parents, not only for the past few years, but also for all the years before. I especially thank Xi, my wife, for all the support that she gave me during the days I have been working on this thesis.

Lastly, I offer my regards and blessings to all of those who supported me in any respect during my doctoral studies.